ANIMATION HOW-TO

See Inside for Animation Contest Details ...

Supes Co

Includes over 450MB of easy-to-use tools, mind-blowing animations, and ready-to-run source code

THE WAITE GROUP®



Bowermaster



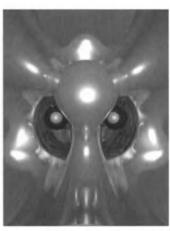


1-878739-54-9



Jeff Bowermaster





HERE'S WHAT YOU CAN WIN:

- First place winner receives \$1000
- Second place winner receives \$500
- Third place winner receives the entire
 Waite Group Presslibrary
- Fourth through tenth place winners receive Waite Group Press T-shirts

See the back of this book for contest details and entry form.

Waite Group Press™ is launching a creative competition to encourage exciting new computerized animations. We're looking for innovative, dramatic, state-of-the-art flics that show off the magic of the PC and simple animation software. You're asked to use the programs in this book. We'll be looking for creativity, spectacular viewpoints, and pizazz. The winners of the contest (1st to 10th) will be presented in an upcoming Waite Group Press animation title.

(C 0)

Jeff Bowermaster



Waite Group Press" Publisher: Mitchell Waite

Editorial Director: Scott Calamar

Managing Editor: John Crudo Content Editor: Harry Henderson Technical Reviewer: David K. Mason Design and Production: Michele Cuneo

Illustrations: Pat Rogondino

Production Director: Julianne Ososke

© 1994 by The Waite Group®, Inc.

Published by Waite Group Press[™], 200 Tamal Plaza, Corte Madera, CA 94925.

Waite Group Press is distributed to bookstores and book wholesalers by Publishers Group West, Box 8843, Emeryville, CA 94662, 1-800-788-3123 (in California 1-510-658-3453).

All rights reserved. No part of this manual shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, desktop publishing, recording, or otherwise, without permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

All terms mentioned in this book that are known to be registered trademarks, trademarks, or service marks are listed below. In addition, terms suspected of being trademarks, registered trademarks, or service marks have been appropriately capitalized. Waite Group Press cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any registered trademark, trademark, or service mark.

The Waite Group is a registered trademark of The Waite Group, Inc. Waite Group Press and The Waite Group logo are trademarks of The Waite Group, Inc. The MTV Logo is a registered trademark of MTV Networks, a division of Viacom International, Inc. © 1993 MTV Networks. All Rights Reserved. Used by permission. Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. QuickBasic is a trademark of Microsoft Corporation. All other product names are trademarks, registered trademarks, or service marks of their respective owners.

MWAVE.FLI and Figure 4-12 © 1992 The Yost Group, used by permission.

Printed in the United States of America 94 95 96 97 • 10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging in Publication Data Bowermaster, Jeff.

Animation how-to CD / Jeff Bowermaster.

p. cm.

Includes index. ISBN: 1-878739-54-9: \$39.95

1. Computer animation. I. Title.

TR897.5.B69 1994

006.6'765--dc20

93-43050

CIP

DEDICATION



o David McCurry, for keeping the wetware stoked while I phased.

ACKNOWLEDGMENTS

he bulk of the credit for the contents goes to the crowd at the YCCMR and TGA BBSs for giving me my education. At the risk of leaving out some important people, here goes: John Hammerton, Dan Farmer, Truman Brown, Mike Miller, Steve Anger, Adam Shiffman, Derek Taylor, Ken Koehler, Doug Reedy, Heinz Schuller, Karl Weller, Chris Smotherman, Douglas Otwell, Jay Sprenkle, John Calcagno, and Jay Schumacher. Their incredible enthusiasm, generosity, and openness provided a nonstop source of inspiration.

Alexander Enzmann and David Mason deserve a round of applause for providing a great ray tracer and a flic builder that are easy, powerful, and for the most part, bug-free. Special thanks go to Eric Deren for being clever, talented, creative, and skilled at uttering the phrase "How's the book coming?"

I'd also sincerely like to thank the folks at Waite Group Press (Mitch Waite, John Crudo, and Michele Cuneo) and the technical talents of Harry Henderson and David Mason for their recommendations and technical expertise.



ABOUT THE AUTHOR

eff Bowermaster got hooked on computer graphics in June 1991 after reading about the DKB ray tracer in *Byte Magazine* and connecting with the folks at the You Can Call Me Ray BBS. He has been animating ray traced images ever since, and has collected a small pile of PCs that double in speed and quadruple in hard disk space each year. He spent eight years racing bicycles and now tortures himself training with those who still do, and lifting weights to make him too heavy to hang on. He has a Ph.D. in Chemistry and works for an agricultural chemical company.

What is a book? Is it perpetually fated to be inky words on a paper page? Or can a book simply be something that inspires—feeding your head with ideas and creativity regardless of the medium? The latter, I believe. That's why I'm always pushing our books to a higher plane; using new technology to reinvent the medium.

I wrote my first book in 1973, Projects in Sights, Sounds, and Sensations. I like to think of it as our first multimedia book. In the years since then, I've learned that people want to experience information, not just passively absorb it—they want interactive MTV in a book. With this in mind, I started my own publishing company and published Master C, a book/disk package that turned the PC into a C language instructor. Then we branched out to computer graphics with Fractal Creations, which included a color poster, 3-D glasses, and a totally rad fractal generator. Ever since, we've included disks and other goodies with most of our books. Virtual Reality Creations is bundled with 3-D Fresnel viewing goggles and Walkthroughs and Flybys CD comes with a multimedia CD-ROM. We've made complex multimedia accessible for any PC user with Ray Tracing Creations, Multimedia Creations, Making Movies on Your PC, Image Lab, and three books on Fractals.

The Waite Group continues to publish innovative multimedia books on cutting-edge topics, and of course the programming books that make up our heritage. Being a programmer myself, I appreciate clear guidance through a tricky OS, so our books come bundled with disks and CDs loaded with code, utilities and custom controls.

By 1993, The Waite Group published 135 books. Our next step is to develop a new type of book, an interactive, multimedia experience involving the reader on many levels.

With this new book, you'll be trained by a computer-based instructor with infinite patience, run a simulation to visualize the topic, play a game that shows you different aspects of the subject, interact with others on-line, and have instant access to a large database on the subject. For traditionalists, there will be a full-color, paper-based book.

In the meantime, they've wired the White House for hi-tech; the information super highway has been proposed; and computers, communication, entertainment, and information are becoming inseparable. To travel in this Digital Age you'll need guidebooks. The Waite Group offers such guidance for the most important software—your mind.

We hope you enjoy this book. For a color catalog, just fill out and send in the Reader Report Card at the back of the book. You can reach me on CIS as 75146,3515, MCI mail as mwaite, and usenet as mitch@well.sf.ca.us.

Mitchell Waite Publisher

Mitchell Waste

Waite Group Press

TABLE OF CONTENTS

Introductio	onxi
Chapter 1	Basics 1
Chapter 2	Fun de Mentals27
Chapter 3	Points of View111
Chapter 4	Mr. Wiggly
Chapter 5	Particle Systems253
Chapter 6	Orchestration
Chapter 7	Blobs
Chapter 8	<i>Recursion</i>
Appendix A	A Tools Reference449
Index	471

Contents

Introduction
Is This Book For You?x
Formatxi
What You Needxi
The Tools
Program Notexi
The READ.ME Filex
Chapter 1 Basics
Preliminaries
General Considerations
Render Time
Looping
Image Size
Frame Count
Antialiasing
Dithering
Aspect Ratio
Basic Examples
1.1 Write a simple scene simulator
1.2 Set a simple scene in motion
Chapter 2 Fun de Mentals
2.1 Move a camera through a scene
2.2 Show a spaceship being launched down a flight tube4
2.3 Use functions to control the positions of objects
2.4 Create a five-dimensional dripping faucet
2.5 Generate a complex surface without resorting to complex triangle grids 9
2.6 Generate tumbling motions
Chapter 3 Points of View
3.1 Create a dazzling background with moving bands of color
3.2 Modify the texture of an object synchronized with the camera motion 11
3.3 Produce a disco inferno

3.4	Generate a spline camera path through a tunnel	134
	Simulate motion down an endless tunnel with a simple model and a few frame	
	Create a bubbling mud-bog surface	
	Randomly materialize an object to show its construction	
	Show cell division and crystal growth	
	Eat away at a complex object to show its interior form	
Chap	ter 4 Mr. Wiggly	179
4.1	Animate ripples on a pond	182
4.2	Make a diamond swim like a jellyfish	188
4.3	Turn a Moravian star inside out	202
4.4	Morph functionally defined objects	206
4.5	Make a manta wave motion with a flapping wedge	211
4.6	Make the letters HMM "happy-train"	215
4.7	Make letters belly dance	226
4.8	Twist a louver ribbon around a bumping three-level blob	230
4.9	Make a rubbery hyperactive cube	240
Chap	ter 5 Particle Systems	253
-	Aminate objects with a free form collision based model	
	Play "red light-green light" with a bunch of spheres	
5.3	Shatter a cube into fragments	267
5.4	Fragment a blob	274
5.5	Give a salt crystal indigestion	279
	Create a swarming swirling twisting smake-like mass that goes on forever	
5.7	Create random numbers in Polyray	301
5.8	Make a volcano erupt with malted milk balls	304
Chap	ter 6 Orchestration	309
	Animate a tumbling buckyball	
	Tumble a group of interlocking rings	
	View color space	
	Do something original with the MTV logo	
	Cover a sphere evenly with other shperes and make it twinkle	
Chan	ter 7 Blobs	359
	Visualize potential fields about objects	
	Create a genetically warped bunch of bananas	

7.3 Tumble blob crosses and see how they form	380
7.4 Resort to udder nonsense	386
7.5 Fly through a soothing fluid tunnel	390
7.6 Fly through tunnels on an asteroid	392
Chapter 8 Recursion	405
8.1 Grow an L-system fan	409
8.2 Generate recursively layered objects	413
8.3 Maximize the use of system memory when building recursive objects	423
8.4 Make a recursive, three-level morphing pat of butter	429
8.5 Make a pentagonal kaleidoscope	443
Appendix A Tools Reference	449
Using QuickBasic	
Using Polyray	
Using DTA	
Using AAPLAYHI	466
Animania Movie Contest	468

INTRODUCTION

ith computer animation using ray traced graphics, you can create anything. Flocks of transparent, skydiving vampire frogs, colliding planets with boiling yellow oceans, elastic spiraling waterfalls, marching columns of pencils with outrageous splurting erasers... anything you can think of, you can create. It's a never ending journey of discovery. It's not like mastering all the commands in some software package and then applying them systematically to solve problems like balancing your checkbook—you get to make all this stuff up as you go along. You wander off into a world of your own creation and decide how you'd like reality to function there. You make up your own rules, create your own tools, set up your own systems and generate whatever you like. You'll no doubt borrow heavily from your experience in this world, but you're not limited by any of its restrictions.

This book describes in detail the process used to generate over 50 ray traced animations. With variations, there are actually over 200 flics, and all the animations are included on the CD-ROM. For a quick preview (and a data rush), a script file has been included to allow you to view all of them at once as soon as you break the seal.

Is This Book For You?

This book chronicles journeys into these other worlds. Its intended audience is people who have some familiarity with computer graphics in addition to basic programming skills. For these people, this book explains powerful procedures and ways of achieving some truly stunning animations and effects.

If you're new to this type of graphics, don't worry. Everything you need to get started is included here. You can pick up the necessary skills by rendering the examples, and reading the code. You can generate some wonderful animations during the learning phase by experimenting to see what certain variables do when given a sequence of different values. Although many people are put off by ray tracing syntax, it's really not that hard to master. It doesn't hurt (much), it sinks in fairly fast, and I'm sure those of you who've already gone through it will agree it's pretty amazing when the bulb clicks on and you realize, "I'm in charge here!"

Where's the GUI?

What might seem strange to many people is that ray tracers (Vivid, Polyray, POV-Ray and Rayshade) are graphics programs without graphical interfaces. It's all text. Scene creation is done by an iterative process of editing, rendering, frowning, and repeating the cycle until a arin is achieved. Motion is added programmatically.

This apparent liability actually turns out to be an asset for most people who stick with it. It forces them to think of scenes mathematically in terms of viewpoints, objects, and lights (see Figure I-1). It's precise, it's controlled, and best of all, it can be automated. Then animation becomes just a matter of herding objects, textures, or indeed any scene descriptor programmatically to create orchestrated effects that would be difficult or at least incredibly tedious to do manually.

600

This book focuses on making scenes move using programs and functions. You should realize that while 3D Studio, Topas, and other high end PC design tools are terrific (if you can afford them), a basic understanding of ray tracing and the principles of scene construction make it possible to create really great scenes without having to spend a dime on the interface. It all comes down to the amount of time you can devote to learning the graphics, versus the amount of convenience you can afford. In a commercial setting, something like 3D Studio can save a great deal of time and effort, but the fundamental effects you create boil down to careful thought and planning, regardless of the tools you use.

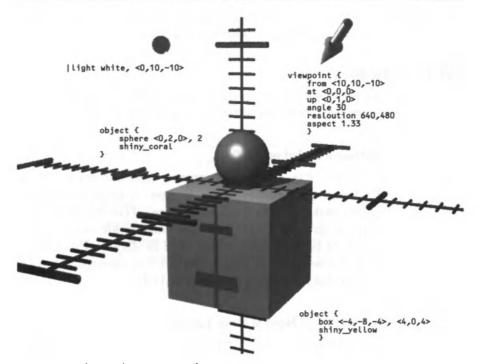


Figure 1-1 Mathematical representation of a scene

Format

This is a hands-on, code-intensive book. The code always preceeds its explanation, but don't let that throw you. Every new concept is covered in detail in the text that follows the code.

Eight chapters contain sample animations that highlight unique effects or principles. An individual animation begins with a description, followed in most cases by a simulation that selects the appropriate values for the control variables. The simulation forms the basis for data files that the ray tracer uses to create the final series of images. Each step in the process is explained in sufficient detail to allow you to employ the principles involved and create animations of your own.

Don't panic. There's no major calculus or cryptic syntax to throw you. No brain cells were harmed in the creation of this code. But this book is by no means what could be described as "content-free." Covered are many useful items such as splines, functions, particle systems, bizarre textures, and motions, with each section focusing in on one particular aspect in the animation process. You'll get the opportunity to play with specific types of animation, and later on combine those principles to make more complex creations. Using the code that was used to create the sample aminations, you'll be able to combine whatever effects you like to create whatever you want.

What You Need

This book and CD, when combined with the proper computer and operating system, provide the resources you need to produce high-end animations.

The Tools Included on the Disc

The Polyray Ray Tracer is the primary rendering engine. It uses simple text files to generate photorealistic images. A series of these images linked together make up your animation. Dave's TGA Animator (DTA) is used to convert a series of images into an animation file called a flic, and can also convert 24-bit targa files, which can't be displayed well by standard VGA adapters, into 8-bit GIF files, which can. The animation player AAPLAYHI is also included for viewing these flics and GIFs.

The Tools You'll Need to Have Yourself

You'll need a decent text editor and Microsoft QuickBasic (which is packaged with MS-DOS 5.0 and above). QuickBasic is used to write simple visualizers, to automate the more tedious aspects of orchestrating complex motions, and to deal with things like several hundred objects at once. Fortunately, QBASIC and EDIT are included with DOS 5.0 (and above).

It wouldn't hurt if you had access to the full QuickBasic package. Editors, like keyboards and word processors, are a matter of personal preference; feel free to use whatever you're comfortable with, but note that an editor with a macro programming language (such as BRIEF or QEDIT) can reduce your work enormously.



Hardware Requirements

Let's not beat around the bush here; ray tracing consumes more raw horsepower than most planet-wide terraforming operations, and carbon-based life forms have annoyingly short life (not to mention attention) spans. Ray tracing is the perfect application to justify getting the newest, fastest, most powerful computer you can find.

The *minimum* system required is a 386 DX with a 387 coprocessor, but a 486 DX or DX2 would be even better. The three-to-one speed advantage of a 486 can be offset by running smaller images on a 386, so you'll still be able to make and display the animations in this book; they'll just either be smaller or take longer to create.

Don't even consider doing this on an SX or a 386 without a coprocessor. Coprocessors are cheap these days and can speed up image rendering 20 times or more. A 486/33 DX with 4MB, an SVGA, a 200MB hard drive, a mouse, and DOS 5.0 or above will get you by quite nicely. The present work was done on three 486s running around the clock for seven months. Memory is usually not a major consideration, and 4MBs is usually fine. It only becomes an issue when you're doing incredibly detailed heightfields or recursive objects. Note, however, that if you have sufficient memory to completely hold your flics in RAM, they play much more smoothly, because the disk transfer bottleneck is removed.

Hardware Wish List

Several items aren't exactly mandatory, but will probably make your life much easier (of course until the bills arrive). You might consider adding them to your wish list for the future or maxing out your credit, depending on your level of self-control.

Hard Drive

A larger hard drive makes dealing with longer animations easier. Flics tend to be huge, but the individual targa files they're built from can be enormous. It's usually a balancing act between blowing away something important ("Do I really need this DOS directory?") and having your animation run out of space sometime around 3 AM.

Tape Backup

A tape backup unit or other removable data storage medium comes in handy to preserve a record of your advancing prowess in computer graphics. QIC-80 tape units are available from several manufacturers, and the ones rated at 250 meg (a specification for compressed text files) will actually store 170-180 meg of flic files on media that goes for under \$20.

24-bit Video Card

The first shocker about VGAs is they aren't perfect. The standard VGA 8-bit mode is a really terrible way to view the wonderful 24-bit images that Polyray generates. Some of you probably have hicolor VGAs or even Windows compatible 24-bit graphics accelerator cards. Using Windows graphics programs like Photoshop, Picture Publisher, Corel Paint, or indeed any Windows image editing program, is highly recommended to show off the true beauty of your images. Be aware that some of the older shareware graphics display programs floating around BBSs might not work well with them.

NTSC Video Output Card

A VGA to NTSC adapter will let you make video tapes and impress your friends. The standard inexpensive ones are not exactly broadcast quality, because there are differences between the aspect ratios and color tolerances of VGA and NTSC displays. But the thrill of seeing your work run to tape is blind to such minor flaws. Just don't expect "mind's eye" quality images.

The Tools

The tools supplied with and required by this book are relatively easy to use; however, you may not be familiar with some of the utilities. Essential information about installing and running these tools is provided here, but it's impossible (in my allotted space) to cover every aspect of their operation. The following introductions into QuickBasic, Polyray, DTA, and AAPLAYHI will get you started, and you'll learn much more about QuickBasic and Polyray as you go through the book's examples, because the code is the heart of the information.

In the allotted space, it's impossible to cover every aspect of QuickBasic use. The following introductions into QuickBasic, Polyray, DTA and AAPLAYHI cover what you'll need to know to get started. You'll end up learning much more about both QuickBasic and Polyray as you go through the examples in each chapter, because the code in this book is where all the real information lies.

The good thing about all this is that the codes are already written and tested, so sit back, relax, maybe grab some popcorn, and enjoy the ride.



Installing the Software

Before we begin to describe the tools, the software provided on the CD-ROM must be installed on your hard drive. Run the installation program, following the directions provided in this book. This will load the programs and utilities into the appropriate directories, ready for use. To find out how to install the provided tools, check out the Installation Guide.

Directories and Drives

Since you may choose to place these files on any drive letter you want, the issue of directory locations for specific files rears its ugly head. Every file in this book assumes you're running Polyray and your data files from the same drive, and demands that these files be contained in directories of a specific name. Provided you're comfortable dealing with directories, you can go in and establish any structure you like.

QuickBasic

Almost every animation created in this book is done with the assistance of QuickBasic. It's great for generating simulations and plotting mathematical functions. It's not particularly powerful or as portable as a language like C, but for what we're using it for, it doesn't have to be. It's easy to write code that runs right the first time.

Loading and Running

Even if you're new to QuickBasic, you'll be able to pick up what's being done here fairly fast. More information on the QBASIC IDE (Intergrated Development Environment) can be found in Appendix A, as well as the DOS documentation. The programs have already been written for you; running them is as simple as typing QBASIC (distributed with DOS) or QB, loading a .BAS file and running it. You may choose to use a mouse to move through the menus, or use the short cut commands to accomplish these actions.

Polyray

The CD contains an installation program that will automatically load the Polyray program, the data files, and all the utilities required into their proper locations.

The shareware version of Polyray includes a batch file that will create all the sample images supplied with Polyray. We didn't do that with the animations in this book because it would take a long time, and require several gigabytes of disk storage space. TGA files tend to form sprawling mounds, and it's useful, at least from a directory listing standpoint, to herd them together into their own subdirectories, use them to create flics, then blow them away.

Running Polyray

Detailed instructions are found in the Polyray documentation which cover all the intricacies of running Polyray and an overview can be found in Appendix A.

DTA

Polyray generates a series of TGA images in a numbered list. DTA assembles them in an animation.

You hand DTA the file prefix for the images, inform it that you want a flic, give it an output name, and a playing speed.

There are many switches in DTA, and please read Appendix A and the program documentation for more information. They're all very useful, but two in particular are used all the time. If you generate images for your flics that are larger than 320 x 200, you must use the /R6 switch to make a high resolution .FLC from them. There are other modes, so again, read the documentation. A quick summary is also available if you just enter the command DTA by itself with no other parameters.

One of the first tasks DTA must perform is generating an optimal palette to map the 16.7 million possible colors in the targa files into the best 256 colors to run on a standard VGA or SVGA display. This can be a lengthy process for animations that are several hundred frames long, and if the basic image doesn't change all that much, checking every single targa file doesn't really make higher quality images. You can scan, say, every 10th image using the /c10 switch, and this can really cut down the time it takes to make the final flic.

DTA generates GIFs using the /fg switch. As mentioned earlier, the animation player AAPLAYHI can then display these GIFs for you, although VPIC or CSHOW, two popular shareware GIF viewers, are more commonly used for this task.

AAPLAYHI

AAPLAYHI is a freely distributable animation player program from Autodesk. The default maximum display size is 320 x 200, but with the appropriate video cards and VESA compatible modes, animations as large as

1,024 x 768 can be displayed. It not only works with flics, but will also display GIFs as well.

An AA.CFG file is created every time AAPLAYHI is started from a directory that doesn't contain one, and the screen size may be selected from whatever screen modes AAPLAYHI detects your hardware is capable of.

If your flic is 320 x 200 or smaller, you can run AAPLAYHI directly by entering the command AAPLAYHI flic.FLI. Entering AAPLAYHI by itself brings up a graphics screen and identifies the program. You press a mouse key to get to the menus. At this point, you'll either be able to load your flics directly or set the screen size to the appropriate one and load your flics afterwords.

AAPLAYHI will use all your system memory and attempt to load the flic into it, which makes flics play much smoother than directly off your hard drive. While it doesn't mind HIMEM, you cannot have EMM386 providing expanded memory or AAPLAYHI will key on it and miss your extended memory entirely.

Additional References

For additional information on the tools in this book, the following resources are available:

- Appendix A, Tools Reference covers QuickBasic, Polyray, DTA, and AAPLATYHI in greater depth.
- The complete documentation for Polyray and DTA on the CD, and your DOS manuals for more information about QuickBasic.
- If you're stumped and need extra help, the companion book Making Movies on Your PC would be a worthwhile resource for you. It provides a complete tutorial on Polyray and DTA by the authors of both programs, and introduces many of the concepts and techniques that we'll be using in this book.
- See installation for more information about loading the tools and code included on the CD.

Program Note

We have used the ← character in this book to indicate program lines that "overflow" onto more than one line. The overflow character means that you should continue to type that line on one line. Do not type the ← symbol or a carriage return.

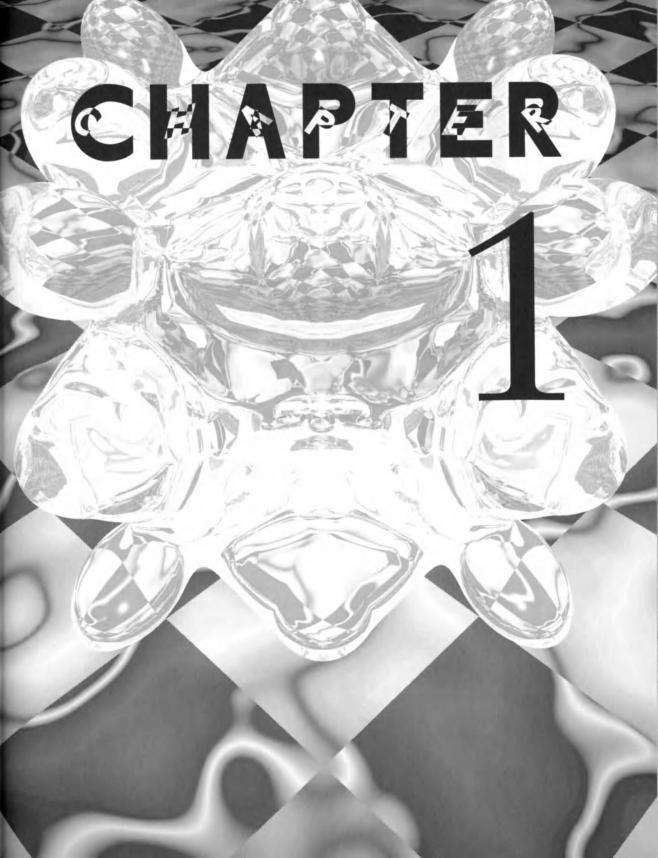
The READ.ME File

Animation How-to CD includes over 450MB of resources, including the final animations, their source code, and the programs described in this book. On the root directory of the CD, you'll find a READ.ME file that explains how the resources are organized, how to use DOS commands to copy what you need to your hard disk, and provides details of any last-minute developments.

You can view the READ.ME file using the DOS TYPE command, as shown in the following example.

type d:\read.me | more

If your CD drive isn't the D: drive, replace d: in the example above with the appropriate letter. A prompt (--MORE--) appears at the bottom of the screen while there is more text for you to read. To scroll ahead, press any key when you're ready to continue.



1

BASICS

hese are wild times. PC horsepower has quietly increased to the point where effects you'd swear would take a high end graphics workstation to create can now be achieved by a person with a PC, a shareware ray tracer and a dream. The trick is translating that dream into code the ray tracer can handle.

Preliminaries

This book deals with creating a variety of ray traced animations using Alexander Enzmann's Polyray ray tracing program. We usually start by simulating the motions involved using QuickBasic, which can both generate graphs showing the time relationships between variables, and generate dynamic wireframe simulations showing the actual motions. We do this in QuickBasic because it's fast and interactive. Doing it inside Polyray would simply take too long. Then we take what we learn from the QuickBasic simulation code and translate that to Polyray code, and generate the final animations either entirely inside Polyray or by using a series of data files that we get QuickBasic to make for us.

General Considerations

The following considerations apply to almost every animation covered in this book, and rather than repeating these concepts over and over in every example, they are gathered here for easy access. While it may be too early to consider these items, be aware of them and their impact. Read them now for general information.

Render Time

Don't be shocked when you find out it might take several days of computer time to make a single animation. This is typical. You only need to consider alternatives when rendering times stretch out to several months, but a couple of days is nothing special. Computers spend most of their time intercepting dust particles anyway. Ray tracers give them a hobby when people aren't sitting in front of them. Every time you leave your computer, start up an animation, turn off the monitor, and come back to see what you've managed to get your computer to make in your absence. It's kind of like sending your-self e-mail (or in this case, e-images) through a time machine.

Looping

Once you've spent hours or days making some five-second animation, you'll probably want to view it over and over again. Flic players do that automatically. A consequence of this is the loop point, where the last frame is followed by the first frame. This can be an important consideration, and may in fact guide how you develop your motions. It's like limericks, in that the form influences the structure. Simple rotations loop nicely, but the variety of possible loops is endless.

Image Size

Most animations in this book are specified at a resolution of 320 x 200, since this size fills the screen with a medium resolution flic. The rendering times are all reasonable, provided you have a fairly fast computer (486/50 or better). If you're working with a 386/387, you might want to drop the image size to 160×100 to cut down your rendering time to something more tolerable for that platform.

The two other important considerations of image size are flic size and display rate. While a 640 x 480 flic might sound nice, a 90 frame animation might take 50 megs of hard disk space for the TGAs and 20 megs for the flic. The rate at which VGAs display information is limited both by the bus speed and by the arcane way VGAs address memory in banks. The end result of these limitations is that full screen, high resolution flics are apt to "tear," meaning that parts of the screen update at different times than other parts, with annoying breaks between these regions. The end result of these shortcomings of the current hardware is that you spend a great deal of time and an enormous amount of disk storage space to generate an image that looks awful when you play it. There's really nothing you can do about these problems except realize the limitations of the hardware and try to accomodate them. It's not necessarily a bad idea to occassionally try out larger images though. A 640 x 480 flic in which only a small portion of the image moves will take less room and display fairly well, since flics store only the differences between frames rather than a sequence of complete images.

Frame Count

It's difficult to guess in advance how many frames you'll need for a particular motion. What you want to do is strike a balance between a flic with jerky motions, and one that's smooth, but huge and ponderous. Something between 10 and 15 frames per second generates smooth motions, although it's not easy arriving at this frame count by looking at your data files. It depends a great deal on the complexity of the images you render. The best way to approach this problem is to start with a few frames, and add more until you achieve the right level of smoothness.

Antialiasing

Antialiasing removes jagged edges from perpendicular objects and generally smooths the overall appearence of your scenes, but it can increase rendering times by a factor of ten! Once you've created some masterpiece and are happy with all aspects of it, you might consider antialiasing for better image quality, but do not use it while you're rendering test frames. You'll become bored.

Dithering

Dithering is a process that extends the range of colors an image can simultaneously display beyond the 256 allotted to a VGA. It does this by scattering pixels of one color inside regions of another color, and gets your eyes to average the two and come up with a third. In effect, you trade spatial resolution for color resolution. DTA includes several dithering options, and for images with slowly varying regions of colors in them, this can really reduce color banding, at the price of decreasing the details and increasing the flic size, in some cases by a factor of two. Dithering was not normally applied to the flics on the CD, and should really be addressed on a case-by-case basis.

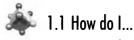
There are several levels of dithering that can be brought to bear on various types of palette problems. Floyd Steinberg dithering produces small dots that move about from frame to frame (accounting for the increased size in such flics), and noise dithering generates irregular patches that remain more or less fixed during playback.

Aspect Ratio

VGAs in 640 x 480 mode have square pixels. The aspect ratio is simply 640/480, or 1.333. VGAs in 320 x 200 mode do not have square pixels. Rendering a sphere in 320 x 200 mode with an aspect ratio of 1.6 (320/200) gives an ellipsoid taller than it is wide. To make matters worse, images rendered at one resolution will appear smashed or stretched when viewed at another. A conscious decision must be made in advance as to what resolution you intend the final work to be viewed. The fastest full-screen mode is 320 x 200, and these images use an aspect ratio of 1.43. A higher quality image can be generated with 320 x 240 run through DTA with the /R6 switch, but this size only covers one quarter of the screen. These images should be generated with an aspect ratio of 1.33.

Basic Examples

We'll start with two simple examples that will involve many of the principles we'll use later on in the book for more complicated animations. The following two animations introduce the concepts of Quickbasic simulations, translation of these simulations to Polyray syntax, and use of the internal animation support in Polyray to create complex nested motion. Remember, information about QuickBasic, Polyray, and the other tools may be found in the Introduction and Appendix A.



Write a simple scene simulator?

You'll find the code for this in: PLY\CHAPTER1\TUMBLE

Problem

The process of rendering a series of images for an animation can be quite time-consuming, particularly when ray tracing is involved. You may not be able to tell if a scene is working until after it's completed, and if something turns out to be wrong, you've wasted all that time. A way around this problem is to create a "quick and dirty" simulation that will show you how your animation will flow before committing major amounts of time to it. This lets you identify and fix problems before you undertake the full rendering process.

In addition to helping you nip errors in the bud, scene simulation allows you to interactively develop structures for complex motions, and to try out new ideas in a fraction of the time it would take a ray tracer to do it, even in wire frame mode. The simulation serves to establish the variables required to manipulate your model. These simulations can be wire frames, simplified scene descriptions using object markers, or something as simple as a series of blinking lights. We'll use QuickBasic code to create our simulations, because it's simple to write and fast enough to handle most models. C would be about five times faster, and it allows more objects, but it can be a pain to use if you're not familiar with it.

Other Options

Most animation packages have some kind of fast previewer built into them, and Polyray is no exception. It has a wire frame rendering mode, which is set by including the line:

renderer wire_frame

This can be added to the POLYRAY.INI file or as a command-line switch. This mode is terrific for checking to see if your scene motion is working and if objects are behaving themselves.

There's also what David Mason calls "postage stamp animations": teeny, blurry animations, something on the order of 32 x 20 pixel images, that simply confirm (in rough terms) that a scene is developing along the right lines. Tiny animations require a lot of squinting and a good deal of imagination.

Both these preview options are viable alternatives to simulation depending on your needs, but for now, let's focus on scene simulation.

Technique

The best-running wire frame previewer uses the following pseudo code loop:

do

```
draw the current scene
save the data for it
calculate the next scene
undraw the current scene (redraw it in black)
```

loop

It typically takes longer to calculate a scene than to display it. The sequence above holds the current image on the screen while the computer is figuring out what the next one looks like. The screen then only blanks for the instant it takes to rip the old image off the screen and slap the new image up onto it.

A word of advice: although it's easier to avoid the bother of saving the current scene for "undrawing" the display (painting it in black) and instead just use the CLS (clearscreen) function, CLS clears every single pixel and takes longer to run than undrawing only those pixels that are lit. CLS results in an annoying screen flicker. So take the time, use arrays and call the current screen points x(n), y(n), and z(n), and write those points to xold(n), yold(n), and zold(n). A cut and paste on the drawing code with a little editing is quick and easy and will give you a very nice simulation in real time with very little flicker.

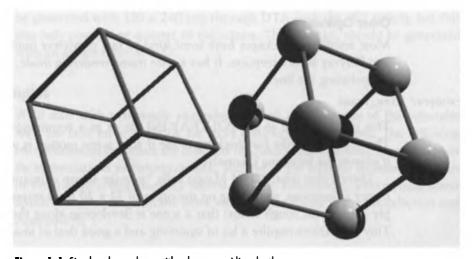


Figure 1-1 Simple cube, and one with spheres providing depth cues

Steps

The following listing, TUMBLE.BAS, creates a very simple tumbling cube wire frame viewer. This wire frame cube uses circles at each corner which get bigger when the side they're on is closest to the viewer. Without such a depth clue, it's hard to tell which side is closer on a wire frame (see Figure 1-1). Indeed, some familiar optical illusions rely on that ambiguity.

```
' TUMBLE.BAS
' Your Basic Tumbling Cube
TYPE Vector
 x AS SINGLE
 y AS SINGLE
 z AS SINGLE
END TYPE
' cube
         = what's displayed
' cubefixed = a static cube as reference
' cubeold = the old screen data
DIM cube(8) AS Vector, cubefixed(8) AS Vector, cubeold(8) AS Vector
SCREEN 12
WINDOW (-3.2, -2.4)-(3.2, 2.4)
pi = 3.1415926535#
rad = pi / 180
' A cube
FOR v = 1 TO 8
       READ cubefixed(v).x, cubefixed(v).y, cubefixed(v).z
NEXT v
' the vertices of a static cube
     x, y, z
DATA 1, 1, 1
DATA 1, 1, -1
DATA 1, -1, 1
DATA 1, -1, -1
DATA -1, 1, 1
DATA -1, 1, -1
DATA -1, -1, 1
DATA -1, -1, -1
DO WHILE INKEY$ = ""
       ' increment an angle (could be frame counter)
       angle = angle + 1
       ' pick some angles to rotate about x,y and z
```

continued on next page

```
continued from previous page
       xrotate = 2 * angle
       yrotate = angle
       zrotate = 90 * SIN(angle * rad)
       FOR a = 1 TO 8
               'calculate the rotation
               ' use fixed cube location
              x0 = cubefixed(a).x
              y0 = cubefixed(a).y
              z0 = cubefixed(a).z
              x1 = x0
              y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
              z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)
              x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
              y2 = y1
              z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)
              x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)
              y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)
              z3 = z2
               ' assign these rotated values to the displayed cube
              cube(a).x = x3
               cube(a).y = y3
               cube(a).z = z3
       NEXT a
       'undraw the old screen
        FOR a = 1 TO 8
              CIRCLE (cubeold(a).x, cubeold(a).y), (cubeold(a).z + 5) / 15, 0
        NEXT a
        LINE (cubeold(1).x, cubeold(1).y)-(cubeold(2).x, cubeold(2).y), 0
        LINE (cubeold(2).x, cubeold(2).y)-(cubeold(4).x, cubeold(4).y), 0
        LINE (cubeold(3).x, cubeold(3).y)-(cubeold(1).x, cubeold(1).y), 0
        LINE (cubeold(4).x, cubeold(4).y)-(cubeold(3).x, cubeold(3).y), 0
        LINE (cubeold(5).x, cubeold(5).y)-(cubeold(6).x, cubeold(6).y), 0
        LINE (cubeold(6).x, cubeold(6).y)-(cubeold(8).x, cubeold(8).y), 0
        LINE (cubeold(8).x, cubeold(8).y)-(cubeold(7).x, cubeold(7).y), 0
        LINE (cubeold(7).x, cubeold(7).y)-(cubeold(5).x, cubeold(5).y), 0
        LINE (cubeold(1).x, cubeold(1).y)-(cubeold(5).x, cubeold(5).y), 0
        LINE (cubeold(2).x, cubeold(2).y)-(cubeold(6).x, cubeold(6).y), O
        LINE (cubeold(3).x, cubeold(3).y)-(cubeold(7).x, cubeold(7).y), 0
        LINE (cubeold(4).x, cubeold(4).y)-(cubeold(8).x, cubeold(8).y), 0
```

```
'draw the new screen
```

```
FOR a = 1 \text{ TO } 8
                CIRCLE (cube(a).x, cube(a).y), (cube(a).z + 5) / 15, a + 1
               ' save the old screen data
                cubeold(a) = cube(a)
        NEXT a
        LINE (cube(1).x, cube(1).y)-(cube(2).x, cube(2).y), 15
        LINE (cube(2).x, cube(2).y)-(cube(4).x, cube(4).y), 15
        LINE (cube(3).x, cube(3).y)-(cube(1).x, cube(1).y), 15
        LINE (cube(4).x, cube(4).y)-(cube(3).x, cube(3).y), 15
        LINE (cube(5).x, cube(5).y)-(cube(6).x, cube(6).y), 15
        LINE (cube(6).x, cube(6).y)-(cube(8).x, cube(8).y), 15
        LINE (cube(8).x, cube(8).y)-(cube(7).x, cube(7).y), 15
        LINE (cube(7).x, cube(7).y)-(cube(5).x, cube(5).y), 15
        LINE (cube(1).x, cube(1).y)-(cube(5).x, cube(5).y), 15
        LINE (cube(2).x, cube(2).y)-(cube(6).x, cube(6).y), 15
        LINE (cube(3).x, cube(3).y)-(cube(7).x, cube(7).y), 15
        LINE (cube(4).x, cube(4).y)-(cube(8).x, cube(8).y), 15
L00P
```

How It Works

DATA statements specify the eight vertices for our cube, and are read into a fixed array (cubefixed(a)) that will serve as a stable reference orientation. All rotations will be made on this reference shape, rather than on each new position the cube would take when it's rotated. This makes the motion easier to loop smoothly.

We then enter a loop where an angle counter is incremented and used to generate (somewhat arbitrarily) three rotational angles about the x, y and z axes:

```
xrotate = 2 * angle
yrotate = angle
zrotate = 90 * SIN(angle * rad)
```

(Don't worry if you don't understand the math here—the main thing is that it works well.) The rotation calculation proceeds to rotate about x, then y, and finally z. Holding variables are required for each axes rotation (x0,y0,z0) since some coordinates changed during the calculations are used again in the same block of code. The cube currently being displayed (cubeold), is "undrawn" by drawing it in black (color 0). The new cube is then drawn on the screen using the color 15 (bright white); we save this new image to cubeold for the next undraw, and the program loops. This cube tumbles until someone touches a key:

```
DO WHILE INKEY$ = ""
```

and INKEY\$ becomes something other than a zero length string ("").

Polyray Version

The Polyray code for the TUMBLE animation is essentially the same as TUMBLE.BAS. In fact it's somewhat simpler, since Polyray has a box primitive and simple vector notation for rotation. Box primitives allow you to call up a box by specifying its opposite corners, instead of having to build six sides with lines. Vector notation makes multidimensional operations as simple to use as addition and subtraction, relieving you of the chore of dealing with each dimension (x, y, and z or R, G, and B) separately. A sample image for the following Polyray data file is shown in Figure 1-2.

```
// A Tumbling Cube with Spheres
start_frame 0
end_frame 71
total_frames 72
outfile "TUMB"
define ang frame* 5
include "PLY\COLORS.INC"
viewpoint {
   from <0,0,-7>
   at <0,0,0>
   up <0,1,0>
   angle 40
   resolution 320,200
   aspect 1.433
background SkyBlue
light white, \langle -5, 5, -10 \rangle
light white, < 5,5,-10>
define pi 3.1415927
define rad pi / 180
define xrotate 2 * ang
define yrotate ang
define zrotate 90 * SIN(ang * rad)
define v1 < 1, 1, 1 > 1
define v2 < 1, 1, -1>
define v3 < 1, -1, 1>
define v4 < 1, -1, -1>
```

```
define v5 <-1, 1, 1>
define v6 <-1, 1, -1>
define v7 <-1, -1, 1>
define v8 <-1, -1, -1>

object {
    object { box <-1,-1,-1>, <1,1,1> matte_white }
    + object { sphere v1, 0.4 shiny_red }
    + object { sphere v2, 0.4 shiny_orange }
    + object { sphere v3, 0.4 shiny_yellow }
    + object { sphere v4, 0.4 shiny_green }
    + object { sphere v5, 0.4 shiny_green }
    + object { sphere v6, 0.4 shiny_blue }
    + object { sphere v6, 0.4 shiny_cyan }
    + object { sphere v7, 0.4 shiny_magenta }
    + object { sphere v8, 0.4 shiny_coral }
    rotate <xrotate,yrotate,zrotate>
}
```

We use 72 frames that multiplied by 5° give 360° of motion during the animation and will return us to our starting point. The eight vertices of a unit cube are defined to specify where the spheres giving us depth cues should be placed, although for the ray traced version it's no longer needed because the cube is solid. A collective object is generated, composed of our box primitive and our spheres, and we rotate the lot with the rotate command.

Comments

QuickBasic simulation code usually translates painlessly to Polyray. Variables like

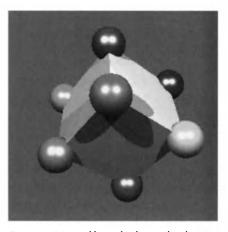


Figure 1-2 A tumbling cube done with Polyray

Note that *angle* is a reserved variable in Polyray, so we must replace it with *ang*. Polyray syntax is sometimes shorter than QuickBasic. The lengthy code for rotation in QuickBasic:

```
x0 = cubefixed(a).x
y0 = cubefixed(a).y
z0 = cubefixed(a).z

x1 = x0
y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)

x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
y2 = y1
z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)

x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)
y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)
z3 = z2

cube(a).x = x3
cube(a).y = y3
cube(a).z = z3
```

condenses down to the single line in Polyray:

rotate xrotate,yrotate,zrotate>

Vector operations like this are incredibly convenient inside Polyray. We'll use them frequently throughout the book.



1.2 How do I ...

Set a simple scene in motion?

You'll find the code for this in: PLY\CHAPTER1\COIN

Problem

Calculating the orientation and position of objects involved in multiple simultaneous motions could be a real nightmare. Something as simple as flipping a coin in the air requires linking a parabolic translation (the toss) to an independent rotation (the flip), and time sequencing the displacements could be a pain. Fortunately, Polyray has the ability to specify an object's motion, along with its shape and color, when it's defined. In fact, any time the object is called, additional motions can be assigned to it that simply add together to produce motions of arbitrary complexity.

This animation shows a metal coin being tossed in the air and it follows a parabolic path. As it flies, it flips about its own axis, revealing both sides. Around the outer perimeter of the coin are six small balls that circulate like the hands of a clock. This demonstrates how to generate motions that are nested three levels deep: (a) the coin flips, (b) the balls circulate, and (c) the whole thing follows a parabolic arc, as shown in Figure 1-3.

Technique

Animating objects requires changing their locations and orientations from frame to frame. While you *could* write separate scene description files for every single frame, Polyray's internal animation support makes it much simpler. Variables may be controlled by the frame counter, which is a variable called *frame* that increments by 1 every frame. A 30 frame animation is specified by including the following lines at the start of the animation:

start_frame 0
end_frame 29
total frames 30

The *frame* variable will assume the values from 0 to 29. You may use this value directly or include it in a formula that in turn generates other values that can be used to control other objects for more complex motions.

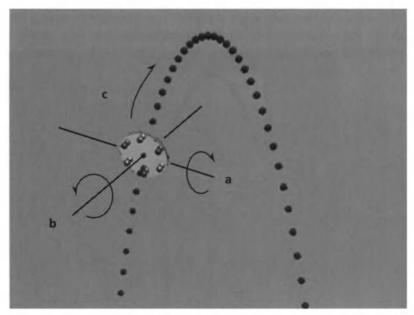


Figure 1-3 Coin flipping geometry

For example, a straight line linear motion of a sphere along the x axis would be:

```
object {
   sphere
   translate <frame, 0 0>
}
```

The frame variable increments once per frame, so this definition becomes

```
translate <1, 0 0>
translate <2, 0 0>
translate <3, 0 0>
```

and so on. To make this motion slower, you'd divide it by some number (e.g. *frame*/10). Then, the object would move 0.1 units each frame. To make our sphere oscillate back and forth along the *x* axis, you'd use *frame* in a sine function:

```
object {
   sphere
   translate <sin(frame/10), 0 0>
}
```

This would cause the position along the x axis to assume the values shown in Figure 1-4.

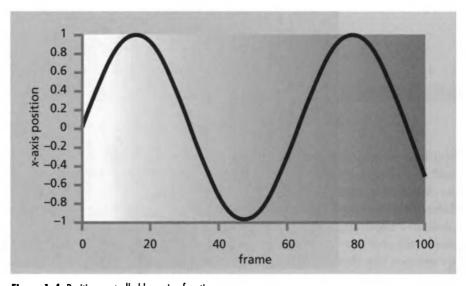


Figure 1-4 Position controlled by a sine function

Complex motions can be predefined in Polyray to keep the object definitions uncluttered:

This would produce a wave form shown in Figure 1-5. If the axis running from -2 to 2 in the figure defined the position of an object down a path, the motion would resemble a typewriter carriage, (jerky forward motion, quick return) for those of you old enough to remember typewriters.

Back to the animation at hand, we have three motions to define. Rotating the balls on the face requires us to define them with the coin and set them in motion prior to flipping the coin, otherwise they won't stick to the surface as it moves. Flipping the coin will then be perpendicular to this rotation. Tossing the coin in the air will be achieved using a parabolic arc, where the effects of gravity slow the upward motion of the coin until it reaches a peak then accelerates back down to earth, with a constant left to right translation across the screen.

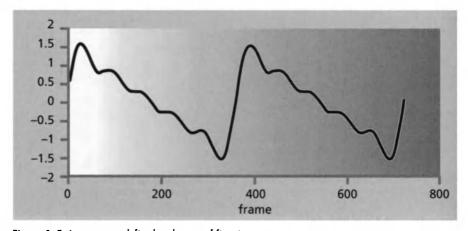


Figure 1-5 A ramp wave defined as the sum of five sine waves

Steps

To create the animation, we need to specify the following items:

- the number of frames
- the file names we want to use for the images (sequentially numbered)
- a viewpoint
- the camera's location
- what it's pointing at
- the resolution of the image
- a background color
- some lights
- include files containing colors and textures we plan to use
- any additional texture definitions not part of the include files
- the objects themselves

COIN.PI, the next listing, shows what these items look like before adding the objects.

```
// COIN.PI
start_frame 0
end_frame 29
total_frames 30
outfile "coin"
viewpoint {
   from <0,6,-16>
   at <0,0,0>
  up <0,1,0>
   angle 45
   resolution 320,200
   aspect 1.433
   }
background <0.439,0.576,0.859>
                                  // a light bluish
light white, < 10,20, -10>
                                    // a white light
include "\PLY\COLORS.INC"
```

```
define copper <0.72,0.45,0.20>

define reflective_copper
texture {
    surface {
        ambient copper, 0.2
        diffuse copper, 0.6
        specular copper, 0.8
        reflection copper, 0.9
        microfacet Phong 10
    }
}
```

The Coin

The coin is constructed from three pieces: a short cylinder capped by two discs, centered on the origin (see Figure 1-6). Cylinders are defined by specifying the centers of both ends and a radius (in this case, the radius is 8 units). Cylinders are not capped, so we must define both faces of the coin as two discs.

Discs are created by specifying their centers, a *surface_normal* (a vector perpendicular to the plane of the disc), and a radius. The next listing collects these objects together into a single object by adding their component parts together.

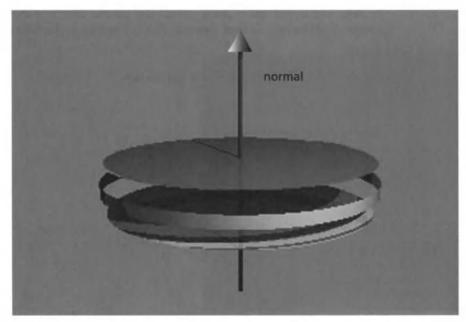


Figure 1-6 Two discs and a cylinder define our coin. Note the surface normals for the discs

```
define flipping_coin
  object {
    object {
        disc <0, 0.5, 0>, <0, 1, 0>, 8
        reflective_copper
    }
    + object {
        disc <0, -0.5, 0>, <0, 1, 0>, 8
        mirror
    }
    + object {
        cylinder <0, -0.5, 0>, <0, 0.5, 0>, 8
        reflective_copper
    }
    rotate <frame*12, 0, 0>
}
```

Circulating Blue Balls

We define an object *blue_balls* that has two counter-rotating spheres embedded in the disks of the coin (Figure 1-7), one on either side. The motion is about the *y* axis, which makes the balls rotate in the horizontal plane of the coin. This motion will stay with the balls regardless of what position the coin is in.

Next we copy this definition six times, with each copy rotated 60° from the last. The object *many_balls*, which distributes them equally around the perimeter of the coin, is then created. This is handled in the following code.

```
define blue_balls
   object {
                  // one side (note the y variable)
      object {
         sphere <6.5,0.5,0>, 1.0
         shiny_blue
         rotate <0, frame*18,0>
      }
                  // the other side
      object {
         sphere <6.5,-0.5,0>, 1.0
         shiny_blue
         rotate <0,-frame*18,0>
      }
   }
define many_balls
   object {
      object {
           blue_balls { rotate <0, 0,0> }
         + blue_balls { rotate <0, 60,0> }
```

```
+ blue_balls { rotate <0,120,0> }
+ blue_balls { rotate <0,180,0> }
+ blue_balls { rotate <0,240,0> }
+ blue_balls { rotate <0,300,0> }
}
```

Now we'll add our *many_balls* definition to the end of the flipping coin definition, as shown here:

```
define flipping_coin
  object {
    object {
        disc <0, 0.5, 0>, <0, 1, 0>, 8
        reflective_copper
    }
    +
    object {
        disc <0, -0.5, 0>, <0, 1, 0>, 8
        mirror
     }
    +
    object {
        cylinder <0, -0.5, 0>, <0, 0.5, 0>, 8
        reflective_copper
     }
    +
    many_balls
        rotate <frame*12,0,0>
}
```

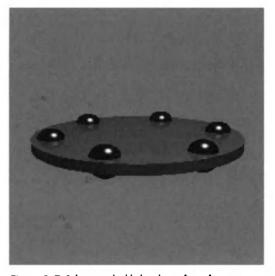


Figure 1-7 Spheres embedded in the surface of our coin

Drawbacks of Direct Frame Counter Control

Before we toss this coin in the air, let's reconsider using the frame counter directly to control its motion. Sure it's convenient, but eventually we'll run into some case where we've hard-coded all the motion to the frame counter and it turns out the speed's all wrong. Recoding every equation containing the frame counter could be tedious. For the present case, doubling the frame count while keeping the motion the same would require replacing *frame*12* with *frame*6* and *frame*18* with *frame*9* (granted, not tough in this case, but in more complicated cases it can become a real mess). It's better to isolate the number of frames in an animation from the actual motion. Then the smoothness can be independently adjusted to whatever is required.

Normalizing the frame counter to a dimensionless number with a fixed range for the entire animation accomplishes this task. Numbers, like the frame counter, have the dimension frame (frame 1, frame 2...). Dividing frame by total_frames makes the number dimensionless. It's then just a number, and in this case, one that goes from 0 to 1 regardless of the number of frames we decide to use.

It's best to pick a range that's convenient for the job at hand. In this coin toss example, we want a parabolic path that peaks halfway through the animation. This motion is easiest to code when the *frame_normal* goes from -1 to 1, because we plan to use an upside down parabola $(y = -(x^2)+c)$ to accomplish this task (Figure 1-8), which reaches its peak when x equals 0.

Angle_normal is another simplification. It's easier to think of the number of times you want something to flip over than the total number of degrees

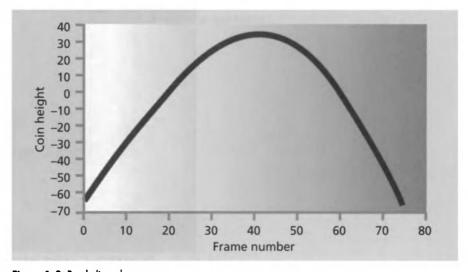


Figure 1-8 Parabolic path

you expect it to rotate. The following listing shows both *frame_normal* and *angle_normal* definitions.

```
start_frame 0
end_frame 99
total_frames 100
define average total_frames/2
//regardless of the total frame count,
//frame_normal always runs -1 to 1
define frame_normal (frame - average)/average
// then angle_normal goes from -180 to 180
define angle normal 180*frame normal
```

Here angle_normal goes from -180° to +180°. Note that if frame_normal ran from 0 to 1, you'd need to define angle_normal as 360*frame_normal, for a total in both cases of 360°. The relationship between the frame counter and these two normals is shown in Figure 1-9.

Tossing the Coin

Now we're ready to toss the thing in the air. Figuring out, based on our viewpoint, where the coin will be visible can either be done with careful planning and graph paper, or simply an educated guess and some wire frame

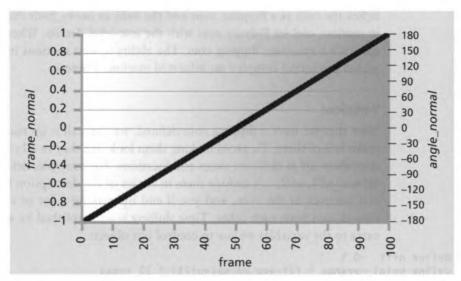


Figure 1-9 The dependency of the values for *frame_normal* and *angle_normal* on the frame counter for a 100 frame animation

rendering. Recall that *wire_frame* is specified in the POLYRAY.INI file or as a command line option. A balance must be reached between being close enough to see the balls rotate on the face of the coin and catching the arc motion as well. The coin will move linearly along the *x* axis and assume an upside down parabolic path along the *y* axis. This is handled as follows:

```
// figured out manually

define ymin -63
define ymax 35
define xmin -35
define xmax 35

define yrange ymax - ymin
define height -yrange * ((frame_normal) ^ 2) +ymax
define dist frame_normal*xmax

object {
    flipping_coin { translate <dist,height,0> }
}
```

Now that the frame count doesn't directly impact the motion, render COIN.PI with *total frames* of 75 and 150 and watch the action.

How It Works

To reiterate the motion-layering concept, if you place an object in motion in its initial definition, when it's called later on, that motion comes along. We define the coin as a *flipping_coin* and the balls as *many_balls* that are already in motion, and let Polyray deal with the low-level details. When we toss the coin, it's a spinning, flipping coin. The ability to nest motions in this manner makes producing complex multifaceted motions a breeze.

Variations

Now that we have a flipping coin defined, we can call it up many times and make lots of them. To avoid making them look like they're all on a rotisserie, start them off at different times and locations. For example, define a series of offsets (off1, off2, ...), include them in copies of the description for the height and position of the coin, and you'll end up making a lot of coins that are time-shifted from each other. Time shifting is accomplished by adding a little extra to the variables we use to control our object:

```
define off1 -0.1
define heig1 -yrange * ((frame_normal+off1) ^ 2) +ymax
define dist1 (frame_normal+off1)*xmax
```

```
define off2 0.1
define heig2 -yrange * ((frame_normal+off2) ^ 2) +ymax
define dist2 (frame_normal+off2)*xmax
define off3 0.3
define heig3 -yrange * ((frame_normal+off3) ^ 2) +ymax
define dist3 (frame_normal+off3)*xmax
define off4 0.5
define heig4 -yrange * ((frame_normal+off4) ^ 2) +ymax
define dist4 (frame_normal+off4)*xmax
define off4 0.7
define heig5 -yrange * ((frame_normal+off5) ^ 2) +ymax
define dist5 (frame_normal+off5)*xmax
object {flipping coin { translate <dist1,heig1,0> }}
object {flipping_coin { translate <dist2,heig2,0> }}
object {flipping_coin { translate <dist3,heig3,0> }}
object {flipping_coin { translate <dist4,heig4,0> }}
object {flipping_coin { translate <dist5,heig5,0> }}
```

Try making one coin solid and fade the others (change their textures to more transparent ones) so that the coin leaves trails. This can also be accomplished by post processing with DTA. Move the balls in and out as the coin rotates. Change their size. Move the camera through the scene to produce depth cues. With appropriate collision detection, a particle systems animation could be written. We'll cover particle systems in Chapter 5, *Particle Systems*.



2

FUN de MENTALS

ere we'll start playing with some general ray traced animations. The first thing we'll probably want to do is fly, since this gets us around fast. Then we'll launch ourselves off into space, build some toys and watch them go, do some incredible ray traced refraction and surface modeling, and end up examining objects, from every angle.



2.1 How do I...

Move a camera through a scene?

You'll find the code for this in: PLY\CHAPTER2\SWOOP

Problem

Animations from a fixed vantage point tend to appear rather flat. A single point of view on a 2-D monitor is like viewing the world with one eye closed. The only depth clues you get are shadows (if there are any), occlusions, and perspective—and only if you know the normal sizes your objects are in relation to each other. Moving the camera around generates differential motion and gives dynamic depth clues about the spatial relationship of objects to other objects in the scene. This produces a much better 3-D feel.

Simple camera motions like linear translations and rotations are easy. We simply use the *frame* variable scaled appropriately to move the camera around as shown here:

```
// linear motion along the x axis
viewpoint {
   from <frame,6,-16>
   at <frame,0,0>
   up <0,1,0>
   angle 45
   resolution 64,48
   aspect 1.33
// orbital motion about the origin, loops every 60 frames
viewpoint {
   from rotate(<6,6,-16>,<0,frame*6,0>)
   at <0,0,0>
   up <0,1,0>
   angle 45
   resolution 64,48
   aspect 1.33
```

More complex motions that follow action or do elaborate aerobatics require more thought and planning. There are many ways to get good gliding motions. We'll deal now with direct functional control, and use splines for more complex motion, in Sections 3.4 and 7.6.

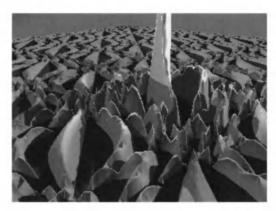


Figure 2-1 The mathematically defined terrain showing a valley we'll thread, next to a mountain

Technique

The current goal is to generate a circling bird's eye view of a mountain in the middle of a plain. The terrain is generated as a heightfield. Heightfields are a wonderfully simple means of generating complex surfaces. We feed Polyray an equation that defines surface height (y) as a function of x and z, and it figures out all the rest. The present surface is defined as a slanting patchwork grid using

```
define HFn (sin(pi * tan(x) + pi * tan(z))) / ra
```

and a mountain is formed in the center by making the height vary inversely with the distance from the origin:

```
define ra (x^2+z^2)^0.5
```

A view of this terrain showing the central mountain and the valley we plan to fly through appears in Figure 2-1. It can be rendered in a pseudo-topographic form (different levels in different colors), using the QuickBasic code shown in TERRY.BAS.

```
' TERRY.BAS
' Rolling Terrain with a Central Mountain

SCREEN 12
WINDOW (-16,-12)-(16,12)
pi = 3.14159
rad = pi/180

FOR radius = .1 TO 12 STEP .1
FOR angle = 1 TO 360

' gimme rings
```

continued on next page

```
continued from previous page
    x = radius * SIN(angle * rad)
    z = radius * COS(angle * rad)

' y is the heightfield
    y = 32 * (SIN(PI * TAN(x) + PI * TAN(z))) / radius

LINE -(x, z), ABS(y + 1) MOD 15
NEXT angle
NEXT radius
```

Steps

Since the surface has been completed, the rest of this animation focuses on defining the components of the path.

The Path

When starting with something as nebulous as a spiraling glide, it's good to write down a description of the kind of motion we're after and then sketch it out on paper (Figure 2-2).

The view begins some distance above the plain, swoops down, circles close to a mountain, flies through the valley next to the central peak, and then swoops back off to where it started. So far then, we've got one order of approaching spiral, a side order of a tight loop, and some connections to

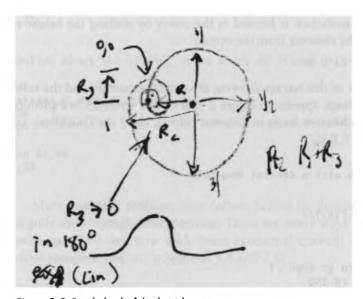


Figure 2-2 Rough sketch of the desired motion

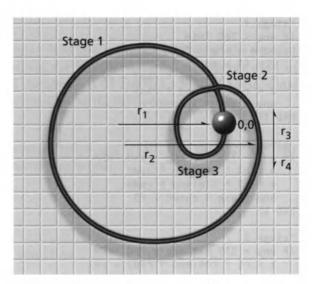


Figure 2-3 The three stages in a spiraling path

make. We'll break it into two components: the spiral path in the x-z plane and the height with respect to y.

The Camera Motion

A more mathematical version of the path we sketched out in Figure 2-2 is shown in Figure 2-3. It starts with radius r1 that increases linearly to r2 through 360°, switches to r3 for the next 180°, then half that (r4) for the remaining 180° to connect back with itself. What we have now is a 720° animation giving us two complete loops $(2 \times 360^\circ)$. There are some x offsets that are required to bring all the arcs together, but they're covered in the code by the expressions -r2 and -r4. The code that draws this path follows.

```
' PATH.BAS
' SPIRAL Path generator
r1 = 12
r2 = 8
r3 = r1 - r2
r4 = r3 / 2
pi = 3.14159
rad = pi/180
speed = 20
DO WHILE ang < 720
```

continued on next page

```
continued from previous page
'stage 1
IF ang < 360 THEN
   norm = ang / 360
   r = (1 - norm) * r2 + norm * r1
   x = r * COS(ang * rad) - r2
   z = r * SIN(ang * rad)
   ang = ang + speed / r
END IF
'stage 2
IF ang > 359 AND ang < 540 THEN
   x = r3 * COS(ang * rad)
   z = r3 * SIN(ang * rad)
   ang = ang + speed / r3
END IF
'stage 3
IF ang > 539 AND ang < 720 THEN
   x = r4 * COS(ang * rad) - r4
   z = r4 * SIN(ang * rad)
   ang = ang + speed / r4
END IF
line -(x,z)
L00P
```

The nice thing about defining the path in this fashion is that r1 and r2 control everything. This allows us to change the relative sizes of both loops and, provided that r1 is greater than r2, everything stays connected. We position the beginning of the r1 loop so that it's centered at the origin; later on we'll adjust it to coincide with a surface feature (the valley in Figure 2-1 of the heightfield), and rotate the whole thing to line up with the orientation of this valley. The result will be the spiral glide over the landscape that we described earlier.

Controlling the Speed

In order to keep the speed of the camera more or less constant at each stage, we index the angle by a variable called *speed* divided by the radius of whatever circle we happen to be in at the time. Recall that radius and circumference are linearly related by the constant 2π . With this division, a smaller radius circle will move the camera through a greater angular displacement than a larger one, but the distance between each of these steps per frame remains constant. Note that this makes figuring out how many frames we'll need to reach the loop point a matter of "run it and count 'em." We get the final frame count from the QuickBasic simulation code.

Swooping Down

Figure 2-4 shows the Gaussian function we'll use to control the camera height. Gaussian functions are the familiar bell-shaped curves used in statistics to model expected distributions of random variables about some average (or center), and are created using the following expression:

height/exp((x-average)/width)^2)

where height sets the maximum value and width sets the spread. This QuickBasic code generates our swoop height:

```
center1 = 230
width1 = 80
height1 = 15

' QB freaks if you don't check first for exponential overflow
' the "IF" expressions avoids these overflows

'gauss 1 for the height
IF ABS(a - center1) / width1 > 7 THEN
    y = 0
ELSE
    y = height1 * (1 / (EXP(((a - center1) / width1) ^ 2)))
END IF
```

Note that in this code, QuickBasic needs to check for exponential overflow; Polyray, due to some nice error trapping, does not.

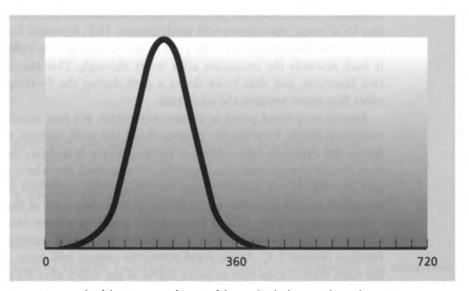


Figure 2-4 Height of the camera as a function of the angular displacement during the animation

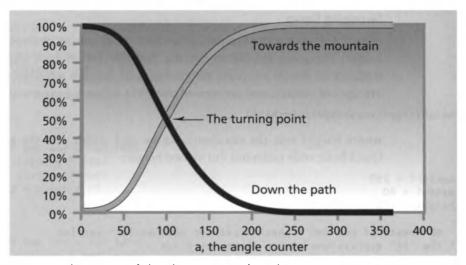


Figure 2-5 The percentage of where the camera points during the animation

Pointing the Camera

The only thing left to deal with is where to point the camera. Since the mountain provides the focal point for the terrain, we'll point the camera at it. Well, actually we'll point it at the valley next to the mountain, since we plan to fly right through the center of it, rather than crash and burn on the mountain. This leaves one problem. As we fly through the valley containing this focal point, the camera will lurch around 180°. It would be better if we could point the camera down the path as we fly through the valley and rotate it back towards the mountain after we're through. This requires blending two functions, one that looks down a path during the fly-through and the other that stares towards the mountain.

Staring at a fixed point is almost automatic; it's just a constant in the viewpoint block. Pointing the camera down the path requires that we save where the camera's been, subtract it from where it is now, then add that vector to its current position. Basically, turn around, see where you've been, add the negative of that to where you are, which gives you where you're going. It's very close to the derivative (tangent to the curve) at that point.

Blending the path with a fixed point requires a time averaging of a constant and a variable. This is accomplished using another Gaussian function (Figure 2-5). At the focal point, the camera will point 100% down the path, then change over to point 100% at the valley after we pass through it. In effect, we're shifting the turning point to somewhere beyond the valley

and spreading the curve out a bit. The blending is accomplished with the following code:

```
center2 = 0
width2 = 100
height2 = 1

' gauss2 for the camera
IF ABS(a - center2) / width2 > 7 THEN
    cam = 0
ELSE
    cam = height2 * (1 / (EXP(((a - center2) / width2) ^ 2)))
END IF

' blend the "at" position
atx = cam * ((x - xold) + x) + (1 - cam) * 0
aty = cam * ((y - yold) + y) + (1 - cam) * 0
atz = cam * ((z - zold) + z) + (1 - cam) * 0
```

Rotating the Path

The valley we're flying through is offset from the origin and runs diagonally across the *x-z* plane. We need to rotate both the camera path and what it's pointed at in order to line it up with the orientation of this valley, then offset them to the center of the valley at frame 1. In the simulation code, we can zoom in and manually adjust the rotation and offset to accomplish this task. Those values are then used in this Polyray code:

```
xrotate = 0
yrotate = 125
zrotate = 0

xfocus = -.5
yfocus = .75
zfocus = -.5
' align the path with the surface features (uses x,y and z rotate)
CALL rotate(x, y, z)

LINE (xold + xfocus, zold + zfocus)-(x + xfocus, z + zfocus), 15
CIRCLE (atx + xfocus, atz + zfocus), .25, 6
```

Visualization Tools

Aligning invisible paths with visible objects is much easier if you make both of them visible. What we'll do is create a solid representation of the path, place it in the scene, step back and render the scene to examine the fit.

Step 1

For a start, place a small sphere at the focal point, pull the camera back to say <1,15,1>, and render the scene. This shows you the focus is where you think it is, as shown in Figure 2-6. (Actually, this shows the path in the next step.)

Step 2

Then, using QuickBasic, generate a file in which the path is defined as a series of little spheres, include this file with the scene file, step back to <20,30,40> and re-render the image. This shows how well your path fits in with the scene (Figure 2-7). Both figures also show the "Look_at" points as a dark path.

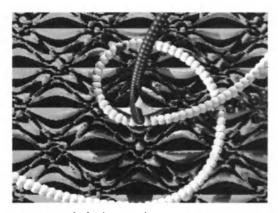


Figure 2-6 The focal point in the animation

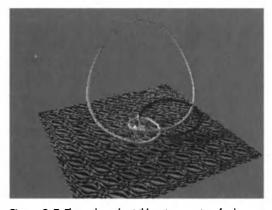


Figure 2-7 The path made visible using a series of spheres

The following QuickBasic visualization program draws the surface, and shows both the camera path and what the camera is pointing at during the fly through. It creates the include file with the path represented by a series of red spheres which get included in our scenes allowing us to render a visible path with the ray tracer.

' PATH.BAS DECLARE SUB rotate (d, e, f) COMMON SHARED rad, xrotate, yrotate, zrotate DIM red(16), green(16), blue(16) SCREEN 12 WINDOW (-24, -18)-(24, 18)FOR y = 1 TO 4 FOR x = 1 TO 4 colornum = x + ((y - 1) * 4) - 1READ red(colornum), green(colornum), blue(colornum) KOLOR = 65536 * blue(colornum) + 256 * green(colornum) + red(colornum) PALETTE colornum, KOLOR COLOR colornum NEXT x NEXT y 'Rainbow Palette 'Rainbow Palette DATA 0, 0, DATA 32, 0, 0 DATA 42, 0, 0 DATA 58, 16, 0 DATA 63, 32, 0 DATA 58, 56, 0 DATA 16, 42, 0 DATA 0, 30, 36 DATA 0, 20, 40 DATA 0, 10, 48 DATA 0, 0, 63 DATA 20, 0, 53 DATA 23, 0, 29 DATA 19, 7, 17 DATA 50, 40, 45 DATA 63, 63, 63 PI = 3.14159rad = PI / 180n = 0

x focus = -.3

continued on next page

CHAPTER TWO

```
continued from previous page
yfocus = .75
zfocus = -.3
FOR radius = .1 TO 15 STEP .1
   FOR angle = 1 TO 360
      x = radius * SIN(angle * rad)
      z = radius * COS(angle * rad)
      y = 32 * (SIN(PI * TAN(x) + PI * TAN(z))) / radius
      IF y < 1 THEN y = 0
      LINE -(x, z), ABS(y + 1) MOD 15
   NEXT angle
NEXT radius
r1 = 12
r2 = 8
speed = 20
r3 = r1 - r2
r4 = r3 / 2
xrotate = 0
yrotate = 125
zrotate = 0
' get these values at the end
xold = -.2587
yold = 0
zold = .21577
center1 = 225
width1 = 80
height1 = 15
center2 = 0
width2 = 100
height2 = 2
LINE (xfocus, zfocus)-(xfocus, zfocus)
OPEN "path.inc" FOR OUTPUT AS #1
DO WHILE a < 720
'stage 1
IF a < 360 THEN
   norm = a / 360
   r = (1 - norm) * r2 + norm * r1
  x = r * COS(a * rad) - r2
   z = r * SIN(a * rad)
   a = a + speed / r
END IF
'stage 2
IF a > 359 AND a < 540 THEN
```

```
x = r3 * COS(a * rad)
  z = r3 * SIN(a * rad)
  a = a + speed / r3
END IF
'stage 3
IF a > 539 AND a < 720 THEN
  x = r4 * COS(a * rad) - r4
  z = r4 * SIN(a * rad)
  a = a + speed / r4
END IF
' align the path with the surface features
CALL rotate(x, y, z)
' QB freaks if you don't check first for exponential overflow
' the "IF" expressions avoids these overflows
'gauss 1 for the height
IF ABS(a - center1) / width1 > 7 THEN
  y = 0
ELSE
  y = height1 * (1 / (EXP(((a - center1) / width1) ^ 2)))
END IF
' gauss2 for the camera
IF ABS(a - center2) / width2 > 7 THEN
  cam = 0
  cam = height2 * (1 / (EXP(((a - center2) / width2) ^ 2)))
END IF
' blend the "at" position
atx = cam * ((x - xold) + x) + (1 - cam) * 0
aty = cam * ((y - yold) + y) + (1 - cam) * 0
atz = cam * ((z - zold) + z) + (1 - cam) * 0
'RAINBOW
LINE (xold + xfocus, zold + zfocus)-(x + xfocus, z + zfocus), y + 1
'WHITE LINE
'LINE (xold + xfocus, zold + zfocus)-(x + xfocus, z + zfocus), 15
CIRCLE (atx + xfocus, atz + zfocus), .1, 6
n = n + 1
'DO WHILE INKEY$ = "": LOOP
PRINT #1, USING "object { sphere < ###.####, ###.#####, ###.####> , 0.20 ←
shiny__red }"; x + xfocus, y + yfocus, z + zfocus
xold = x
yold = y
zold = z
L00P
```

continued on next page

```
continued from previous page
PRINT "frames = "; n; " final angle = "; a - speed / r4
PRINT USING "final location <###.#####, ###.#####, ###.####>"; x, y, z
CLOSE #1
SUB rotate (d, e, f)
    x0 = d
    y0 = e
    z0 = f
    x1 = x0
    y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
    z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)
    x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
    y2 = y1
    z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)
    x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)
    y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)
    z3 = z2
    d = x3
    e = y3
    f = z3
```

Step 3

END SUB

To use the include file this program generates, add the following lines to the Polyray listing in the next section:

```
light white*0.5, <20,35,40>
include "path.inc"
```

Then add the two fixed vantage points—one for the whole path, the other for focusing on the valley—with the following viewpoint code.

```
// set up the camera
viewpoint {
   from <20,30,40> // to see the whole path
   at <0,5,0>
// from <1,15,1> // to see the valley focal point
// at <0,5,0>
   up <0,1,0>
   angle 35
   resolution 320,240
   aspect 1.33
```

Polyray Data File

The Polyray translation follows pretty much the same style as the QuickBasic simulation code shown here in SWOOP.PI.

```
// SWOOP.PI
// Tangent Pattern Flyover
start_frame O
end frame 231
total_frames 232
outfile swoop
define pi 3.1415927
define rad pi/180
define r1 12
define r2 8
define r3 r1 - r2
define r4 r3/2
define speed 20
define align <0,125,0>
define focus <-0.3, 0.75, -0.3>
define origin <0,0,0>
if (frame==start_frame) {
   static define a O
   static define was_at rotate(<-0.2587,0,0.21577>,-align)
   static define fromvect focus
7
// stage 1
if (a < 360) {
   define norm a / 360
   define r (1 - norm) * r2 + norm * r1
   define x1 r * COS(a * rad) - r2
   define z1 r * SIN(a * rad)
   static define a a + speed / r
}
// stage 2
if (a>359 && a< 540) {
   define x1 r3 * COS(a * rad)
   define z1 r3 * SIN(a * rad)
   static define a a + speed / r3
// stage 3
```

continued on next page

```
continued from previous page
if (a >539 && a <720) {
   define x1 r4 * COS(a * rad) - r4
   define z1 r4 * SIN(a * rad)
   static define a a + speed / r4
}
define center1 240
define width1 80
define height1 15
define center2 0
define width2 100
define height2 2
define y1 height1 * (1 / (EXP(((a - center1) / width1) ^ 2)))
define cam height2 * (1 / (EXP(((a - center2) / width2) ^ 2)))
static define fromvect <x1,y1,z1>
define atvect (cam * ((fromvect-was_at) + fromvect) + (1 - cam) * origin)
static define was_at fromvect
// Set up the camera
viewpoint {
   from rotate(fromvect,align)+focus
   at rotate(atvect,align)+focus
   up <0,1,0>
   angle 35
   resolution 320,240
   aspect 1.33
}
// Get various surface finishes
include "\ply\colors.inc"
define mountain_colors
texture {
   noise surface {
      ambient 0.25
      diffuse 0.8
      specular 0.2
      position_fn 1
      color map(
         [-1.28,-0.66, MidnightBlue, Navy]
         [-0.66,-0.30, Navy, Blue]
         [-0.30, 0.00, Blue, MediumBlue]
         [ 0.00, 0.20, ForestGreen, SpringGreen]
         [ 0.20, 0.40, SpringGreen, Gold]
         [ 0.40, 0.80, Gold, GoldenRod]
         [ 0.80, 1.28, GoldenRod, Gray])
   rotate <0, 0, 90>
```

```
// Set up background color & lights
background MidnightBlue
spot_light <1.0,1.0,1.0>, <-20,10,-20>, <0,0,0>, 3, 25, 45
spot_light <1.0,1.0,1.0>, <20,10,-20>, <0,0,0>, 3, 25, 45
define ra (x^2+z^2)^0.5
define HFn (sin(pi * tan(x) + pi * tan(z))) / ra
define detail 200
// Define a patterned surface
   object {
      smooth_height_fn detail, detail, -15, 15, -15, 15, HFn
      mountain colors
   }
// The following code was used to view the path
// remove the comment slashes (//) to use it
// Set up the camera
//viewpoint {
     from \langle 20, 30, -40 \rangle // to see the whole path
     at <0,5,0>
       from <1,15,1> // to see the valley focal point
1111
1111
       at <0,5,0>
//
     up <0,1,0>
//
     angle 35
//
     resolution 640,480
//
     aspect 1.33
11
     }
//include "path.inc"
//light white, <20,35,40>
```

How It Works

The frame count goes from 0 to 231, but the motivating variable is actually *a*, the angle count, which goes from 0 to 720. Polyray uses the key word *static* to define variables whose values stick around from frame to frame, allowing us to build upon their previous values to create subsequent ones.

There are three stages in the Polyray file, as in the QuickBasic code, that generate the spiraling path of various radii under control of the angle counter a. The two Gaussian functions specify y1, the height of the path above the surface as a function of the angle, and cam, the proportioning of where the camera is pointing, between a fixed viewpoint and one that is moving down the path. The fixed viewpoint is the origin; the one moving down the path is given by:

```
((fromvect-was_at) + fromvect)
```

which adds the current path direction to the current camera location to generate a view that follows the path. These two are proportionally added depending on the *cam* value to generate the *atvect*.

```
define atvect (cam * ((fromvect-was_at) + fromvect) + (1 - cam) * origin)
```

These variables *fromvect* and *atvect* specify the camera position and what the camera looks at. Next, the path must be rotated to align it with the surface features, which is done "on the fly" inside the viewpoint code. After the rotation, the entire path is offset by the vector *focus* to move it into the center of the valley on the surface.

After calculating fromvect and atvect, was_at is set to fromvect to be used in the next frame to generate the current camera path. This brings up one issue. How is the first frame variable was_at determined? It's done manually in the simulation code.

```
if (frame==start_frame) {
   static define a 0
   static define was_at rotate(<-0.2587,0,0.21577>,-align)
   static define fromvect focus
}
```

We run the entire simulation, and print out the coordinates of the camera in the last frame. The major difference between the QuickBasic code and the Polyray code is that QuickBasic rotates the path on the fly, while the Polyray code rotates the path at the end, in the viewpoint block. The last PRINT statement in the simulation code gives the rotated value for the last position of the camera. Since the animation loops, this is the was_at value for the first frame, but before Polyray can use it, we must de-rotate it by the angle -align.

Mountain_colors is a texture map using a linear gradient ranging from -1.28 to 1.28, which varies from blue to green to tan to white. Color maps default to running left to right in the x-z plane. This map needs to be tipped on its side using a 90° rotation to make it run up and down, which allows us to highlight variations in surface height with different colors.

The smooth heightfield has a variable *detail* that allows us to specify how detailed we want the heightfield to be. More detail takes more memory, more time, and can dramatically change the look of a surface as compared to a less detailed surface, since the heightfield function is not antialiased. The heightfield function does not use the average value for the controlling function over an area, but rather the value at a specific point on the surface. If the function defining the surface has elements whose dimensions are smaller than the sampling grid defined by detail, spurious details may appear. The height and exact position of the foothills and the mountain change depending on the level of detail that is used.

Comments

For all the effort we went through to smooth the focal point transition during the flythrough, we really only moved it away from the center of the valley. There's still a pretty substantial lurch as the camera swings around to point back at the mountain. A shock absorber approach, one that would limit the rate of change in the angular rotation of the camera to some maximum value, would probably do a better job. It requires keeping track of the angular change, an error term between where the code wants the camera to point and where it's actually pointing, and an if-else construction to select between the two. This animation is already complicated enough, however, so we'll leave it as it is for now. Don't let this stop you from trying it.



🎉 2.2 How do I...

Show a spaceship being launched down a flight tube?

You'll find the code for this in: PLY\CHAPTER2\ROCKET

Problem

Animations involving long linear camera motions are expensive from a modeling standpoint, since all the work required to generate something interesting in front of the camera is lost as soon as the camera passes by the objects. We may only get to see the entire model at the beginning of the animation. For repetitive structures like hallways or tunnels, we can move parts of the model back up in front of the camera after we pass them. This makes long structures appear to go on forever. It's known as *treadmilling*.

Technique

We'll create a simulation of the animation without treadmilling in order to examine timing and frame count, do a Polyray animation from this information, and then add treadmilling and chaser lights. The basic concept of this animation is to watch a rocket pass closely by the camera and fly out the end of a launch tube.

Steps

We build a tunnel from the intersection of crossed cylinders, and trolley the camera from one end of the tunnel to the other. Halfway through the animation, a rocket ship launched from some distance behind the camera at

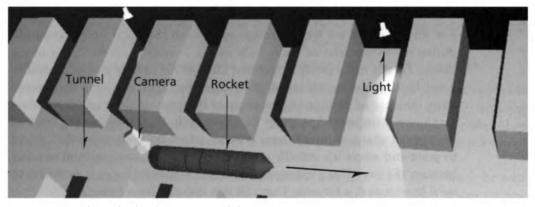


Figure 2-8 Launch tunnel with rocket, camera, and light source

twice its speed overtakes it and we watch as the rocket majestically passes and flies out the end of the tunnel (Figure 2-8).

We can simulate this motion with the following QuickBasic code. It shows an outside view of the action to give us a feel for the timing and choreography.

```
' ROCKET1.BAS
' Polyray Rocket Launch Animation Simulation
SCREEN 12
WINDOW (-32, -24)-(32, 24)
'the tunnel - alternating tall and square boxes
FOR x = -15 TO 16
   IF x \text{ MOD } 2 = 0 \text{ THEN}
      LINE (x - .5, -2)-(x + .5, 2), 15, B
   ELSE
      LINE (x - .5, -.5)-(x + .5, .5), 15, B
   END IF
NEXT x
   some constants - they stay fixed but follow the
   syntax of the changing variables
cameray = .2 ' camera y-offset
lookaty = 0 ' look at y-offset rockety = 0 ' rocket y-offset
ocy = cameray ' old camera y-offset
oly = lookaty ' old look at y-offest
ory = rockety ' old rocket y-offset
totalframes = 200
FOR frame = 0 TO totalframes
   norm = frame / totalframes ' frame normal - 0 to 1
```

```
cameraz = -15 + 30 * norm ' camera goes from -15 to 15
   lookatz = -10 + 30 * norm ' look at leads the camera by 5 rocketz = -30 + 60 * norm ' rocket goes from -30 to 30
   'undraw
   FOR section = -4 TO 4
      LINE (orz - .4 + section, ory - .05)-
      (orz + .4 + section, ory + .05), 0, BF
   NEXT section
   CIRCLE (ocz, ocy), .2, 0
   CIRCLE (olz, oly), .2, 0
   'draw
   FOR section = -4 TO 4
      LINE (rocketz - .4 + section, rockety - .05)-
      (rocketz + .4 + section, rockety + .05), 4, BF
   NEXT section
   CIRCLE (cameraz, cameray), .2, 4
   CIRCLE (lookatz, lookaty), .2, 2
   'save
   orz = rocketz
   ocz = cameraz
   olz = lookatz
   ' a pause loop to slow it down for viewing
   FOR w = 1 TO 10000: NEXT w
NEXT frame
```

The Polyray code gives us the view from the camera's perspective. The launch tunnel is comprised of 33 short cylinders staggered at right angles to the flight path. A central cylinder is drilled through all of their centers to define the flight tube (Figure 2-9).

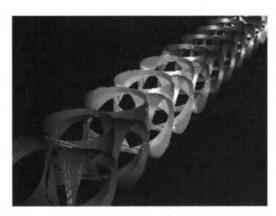


Figure 2-9 Our rocket launch tunnel

ROCK1.PI, the next listing, builds the rocket out of nine cylinders. Small spaces defined by the variable *slot* separate the cylinders. The rocket passes over a light in the center of the tunnel, and the light that leaks out through the slots shows up as stripes on the walls of the launch tube.

```
// ROCK1.PI
start_frame 0
end_frame 150
total_frames 90 // don't worry about this.
outfile rock
include "\PLY\COLORS.INC"
define norm frame/total frames
define rocketz -40 + frame/3
viewpoint {
   from <0.20, 0.20, -14+14*norm>
   up
         <0, 1, 0>
         <0,0.0,-10+14*norm>
   at
   angle 25
   resolution 160,120
   aspect 1.33
// the red rocket
define slot 0.47
define section
   object {
      cylinder <0,0,-0.5+slot>,<0,0, 0.5-slot>,0.2
define t1 <0,0,-4+rocketz>
define t2 <0,0,-3+rocketz>
define t3 <0,0,-2+rocketz>
define t4 <0,0,-1+rocketz>
define t5 <0,0, 0+rocketz>
define t6 <0,0, 1+rocketz>
define t7 <0,0, 2+rocketz>
define t8 <0,0, 3+rocketz>
define t9 <0,0, 4+rocketz>
define rocket
object {
     object { section translate t1 }
   + object { section translate t2 }
   + object { section translate t3 }
   + object { section translate t4 }
   + object { section translate t5 }
   + object { section translate t6 }
   + object { section translate t7 }
```

```
+ object { section translate t8 }
  + object { section translate t9 }
  + object { cone <0, 0, 0>, 0.2, <0, 0, 1>, 0 translate t9+<0,0,0.5>}
 shiny_red
light <0,0,0>
light <-5, 0, 10>
light < 5, 0, 10 >
background midnightblue
define launch_tube
  object {
// the side chambers
     object { cylinder <-2, 0,-16>, < 2, 0,-16>, 1 }
   + object { cylinder < 0,-2,-15>, < 0, 2,-15>, 1 }
   + object { cylinder <-2, 0,-14>, < 2, 0,-14>, 1 }
   + object { cylinder < 0,-2,-13>, < 0, 2,-13>, 1 }
   + object { cylinder <-2, 0,-12>, < 2, 0,-12>, 1 }
   + object { cylinder < 0,-2,-11>, < 0, 2,-11>, 1 }
   + object { cylinder <-2, 0,-10>, < 2, 0,-10>, 1 }
   + object { cylinder < 0,-2, -9>, < 0, 2, -9>, 1 }
   + object { cylinder <-2, 0, -8>, < 2, 0, -8>, 1 }
   + object { cylinder < 0,-2,-7>, < 0, 2,-7>, 1 }
   + object { cylinder <-2, 0, -6>, < 2, 0, -6>, 1 }
   + object { cylinder < 0,-2,-5>, < 0, 2,-5>, 1 }
   + object { cylinder <-2, 0, -4>, < 2, 0, -4>, 1 }
   + object { cylinder < 0,-2, -3>, < 0, 2, -3>, 1 }
   + object { cylinder <-2, 0, -2>, < 2, 0, -2>, 1 }
   + object { cylinder < 0,-2, 1>, < 0, 2, 1>, 1 }
   + object { cylinder <-2, 0, 0>, < 2, 0,
   + object { cylinder < 0,-2, 1>, < 0, 2,
                                             1>, 1 }
   + object { cylinder <-2, 0, 2>, < 2, 0, 2>, 1 }
   + object { cylinder < 0,-2, 3>, < 0, 2, 3>, 1 }
   + object { cylinder <-2, 0, 4>, < 2, 0, 4>, 1 }
   + object { cylinder < 0,-2, 5>, < 0, 2, 5>, 1 }
   + object { cylinder <-2, 0, 6>, < 2, 0,
                                            6>, 1 }
   + object { cylinder < 0,-2, 7>, < 0, 2, 7>, 1 }
   + object { cylinder <-2, 0, 8>, < 2, 0, 8>, 1 }
   + object { cylinder < 0,-2, 9>, < 0, 2,
                                             9>, 1 }
   + object { cylinder <-2, 0, 10>, < 2, 0, 10>, 1 }
   + object { cylinder < 0,-2, 11>, < 0, 2, 11>, 1 }
   + object { cylinder <-2, 0, 12>, < 2, 0, 12>, 1 }
   + object { cylinder < 0,-2, 13>, < 0, 2, 13>, 1 }
   + object { cylinder <-2, 0, 14>, < 2, 0, 14>, 1 }
   + object { cylinder < 0,-2, 15>, < 0, 2, 15>, 1 }
   + object { cylinder <-2, 0, 16>, < 2, 0, 16>, 1 }
   & ~object {cylinder < 0, 0,-18>, < 0, 0, 18>, 1}
   texture {surface {color coral ambient 0.3 diffuse 0.8}}
}
launch_tube
```

Treadmilling the Model

In order to treadmill the cylindrical sections of the launch tube, we define a variable called *nudge* that shifts the entire tunnel towards the camera a little bit each frame. It uses modulo math, that is, the remainder of one number divided by another number. Since the other number is 20, it repeats every 20 frames. With the repeating units of the tunnel being two units apart, and the fact that we're dividing the modulo term by 10, the tunnel also lines up with its original orientation every 20 frames. Here's how:

```
define nudge 2 - fmod(frame,20) / 10
define launch_tube
  object {
// The side chambers
    object { cylinder <-1.2, 0,-18+nudge>, < 1.2, 0,-18+nudge>, 1 }
  + object { cylinder < 0,-1.2,-17+nudge>, < 0, 1.2,-17+nudge>, 1 }
  + object { cylinder <-1.2, 0,-16+nudge>, < 1.2, 0,-16+nudge>, 1 }
```

This dramatically changes the way we generate the apparent motion. Before, we moved the camera down the tunnel. Now we'll hold the camera steady, move the tunnel towards us and the rocket away from us, but from the camera's perspective, it looks exactly the same as the previous animation. The main difference is that the tunnel appears to go on forever, while the model is in fact of finite length, with a (better still) finite rendering time.

Chaser Lights

Although the light leaking through the slots in the rocket in the previous animation was a nice effect, it would fail to seriously stun anyone. The desired effect was much more dramatic: moving lights tugging our rocket forward like the powerful magnetic forces in a linear accelerator.

Let's use chasers. These are a series of moving lights that travel down the length of the tube, as shown by lines in Figure 2-10. The lines actually represent four spotlights surrounding the launch tube, shining through the holes at an angle.

ROCKETZ.BAS, shown in the following code, shows the chaser lights as they relate to the launch tube. Remember, the tube's being pulled one way, the rocket's moving out the other way, and the camera's staying put. The ends of this simulation are a bit rough, but it gives you a pretty good idea of the motion. Use modulo math again to get a linear displacement that repeats, in this case every 16 frames.

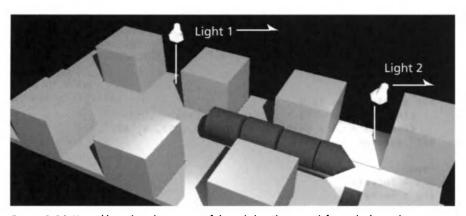


Figure 2-10 Vertical lines show the positions of chaser lights. They move left to right during the animation

```
' ROCKETZ.BAS
SCREEN 12
WINDOW (-32, -24)-(32, 24)
' a tunnel conveyor belt
FOR frame = 0 TO 200
  nudge = 2 - (frame MOD 20) / 10
  LOCATE 10, 38: PRINT USING "### "; frame
' pause code-use as needed
   FOR w = 1 TO 10000: NEXT w
   'the tunnel
   FOR px = -15 TO 16
   'undraw
      ox = px + nudge + .1
      IF px MOD 2 = 0 THEN
         LINE (ox - .5, -2)-(ox + .5, 2), 0, B
         LINE (ox - .5, -.5)-(ox + .5, .5), 0, B
      END IF
   'draw
      x = px + nudge
      IF px MOD 2 = 0 THEN
         LINE (x - .5, -2)-(x + .5, 2), 15, B
      ELSE
         LINE (x - .5, -.5)-(x + .5, .5), 15, B
      END IF
  NEXT px
   'the ship
   rocketx = -40 + frame / 3
   'undraw
```

```
continued from previous page
  FOR section = -4 TO 4
     LINE (orx -.4 + section, ory -.05)-
     (orx + .4 + section, ory + .05), 0, BF
  NEXT section
  'draw
  FOR section = -4 TO 4
     LINE (rocketx - .4 + section, rockety - .05)-
     (rocketx + .4 + section, rockety + .05), 4, BF
  NEXT section
  'save
  orx = rocketx
  ory = rockety
  'traveling "chaser" lights using modulo math again
  trl1 = -16 + frame MOD 16
  trl2 = 0 + frame MOD 16
  LINE (trl1, -5)-(trl1, 5), count
  LINE (trl2, -5)-(trl2, 5), count
  IF trl2 = 0 THEN count = count + 1
NEXT frame
```

Polyray Data File

The final animation sequence (ROCKET.PI) can either hold the camera steady inside the tube, or shift from the outside to the inside of the launch tube just to show off how good the tunnel looks from the outside. This second animation doesn't loop smoothly, so that code (the lines following the comment "shifting point of view") has been commented out, but you may wish to try it out. It's an exponential shift between two viewpoints v0 and v1.

```
// ROCKET.PI - Rocket Launching Animation
start_frame 0
end_frame 160
total_frames 160
outfile rock
include "..\colors.inc"
// shifting point of view - next 4 lines
```

```
// define v0 <10,10,-30>
// define v1 <0.45,0.35,-16>
// define shift 1/exp(frame/16)
// define v2 shift*v0+(1-shift)*v1
// steady point of view
   define v2 <0.45,0.35,-16>
viewpoint {
   from v2
   at <0,0,-12>
   up <0, 1, 0>
   angle 25
   resolution 320,200
   aspect 1.33
   }
// The ROCKET Definition Code
define slot 0.02
define section
   object {
      cylinder <0,0,-0.5+slot>,<0,0, 0.5-slot>,0.2
      }
define rocketz -40 + frame / 3
define t1 <0,0,-4+rocketz>
define t2 <0,0,-3+rocketz>
define t3 <0,0,-2+rocketz>
define t4 <0,0,-1+rocketz>
define t5 <0,0, 0+rocketz>
define t6 <0,0, 1+rocketz>
define t7 <0,0, 2+rocketz>
define t8 <0,0, 3+rocketz>
define t9 <0,0, 4+rocketz>
define rocket
object {
     object { section translate t1 }
   + object { section translate t2 }
   + object { section translate t3 }
   + object { section translate t4 }
   + object { section translate t5 }
   + object { section translate t6 }
   + object { section translate t7 }
   + object { section translate t8 }
   + object { section translate t9 }
   + object { cone <0, 0, 0>, 0.2, <0, 0, 1>, 0 translate t9+<0,0,0.5>}
 shiny_red
 }
rocket
```

CHAPTER TWO

```
continued from previous page
// engine inside rocket
light <1,1,2>,t1
object {
   sphere t1,0.2
   shading_flags 32+8+4+2+1
}
//light t2
//light t3
//light t4
light t5
//light t6
//light t7
//light t8
light t9
background midnightblue
define nudge 2 - fmod(frame, 20) / 10
define launch_tube
   object {
// The side chambers
      object { cylinder <-1.2, 0,-18+nudge>, < 1.2, 0,-18+nudge>, 1 }
    + object { cylinder < 0,-1.2,-17+nudge>, < 0, 1.2,-17+nudge>, 1 }
    + object { cylinder <-1.2, 0,-16+nudge>, < 1.2, 0,-16+nudge>, 1 }
    + object { cylinder < 0,-1.2,-15+nudge>, < 0, 1.2,-15+nudge>, 1 }
    + object { cylinder <-1.2, 0,-14+nudge>, < 1.2, 0,-14+nudge>, 1 }
    + object { cylinder < 0,-1.2,-13+nudge>, < 0, 1.2,-13+nudge>, 1 }
    + object { cylinder <-1.2, 0,-12+nudge>, < 1.2, 0,-12+nudge>, 1 }
    + object { cylinder < 0,-1.2,-11+nudge>, < 0, 1.2,-11+nudge>, 1 }
    + object { cylinder <-1.2, 0,-10+nudge>, < 1.2, 0,-10+nudge>, 1 }
    + object { cylinder < 0,-1.2, -9+nudge>, < 0, 1.2, -9+nudge>, 1 }
    + object { cylinder <-1.2, 0, -8+nudge>, < 1.2, 0, -8+nudge>, 1 }
    + object { cylinder < 0,-1.2, -7+nudge>, < 0, 1.2, -7+nudge>, 1 }
    + object { cylinder <-1.2, 0, -6+nudge>, < 1.2, 0, -6+nudge>, 1 }
    + object { cylinder < 0,-1.2, -5+nudge>, < 0, 1.2, -5+nudge>, 1 }
    + object { cylinder <-1.2, 0, -4+nudge>, < 1.2, 0, -4+nudge>, 1 }
    + object { cylinder < 0,-1.2, -3+nudge>, < 0, 1.2, -3+nudge>, 1 }
    + object { cylinder <-1.2, 0, -2+nudge>, < 1.2, 0, -2+nudge>, 1 }
    + object { cylinder < 0,-1.2, 1+nudge>, < 0, 1.2, 1+nudge>, 1 }
    + object { cylinder <-1.2, 0, 0+nudge>, < 1.2, 0, 0+nudge>, 1 }
    + object { cylinder < 0,-1.2, 1+nudge>, < 0, 1.2, 1+nudge>, 1 }
    + object { cylinder <-1.2, 0, 2+nudge>, < 1.2, 0, 2+nudge>, 1 }
    + object { cylinder < 0,-1.2, 3+nudge>, < 0, 1.2, 3+nudge>, 1 }
    + object { cylinder <-1.2, 0, 4+nudge>, < 1.2, 0, 4+nudge>, 1 }
    + object { cylinder < 0,-1.2, 5+nudge>, < 0, 1.2, 5+nudge>, 1 }
    + object { cylinder <-1.2, 0, 6+nudge>, < 1.2, 0, 6+nudge>, 1 }
    + object { cylinder < 0,-1.2, 7+nudge>, < 0, 1.2, 7+nudge>, 1 }
    + object { cylinder <-1.2, 0, 8+nudge>, < 1.2, 0, 8+nudge>, 1 }
    + object { cylinder < 0,-1.2, 9+nudge>, < 0, 1.2, 9+nudge>, 1 }
    + object { cylinder <-1.2, 0, 10+nudge>, < 1.2, 0, 10+nudge>, 1 }
    + object { cylinder < 0,-1.2, 11+nudge>, < 0, 1.2, 11+nudge>, 1 }
```

```
+ object { cylinder <-1.2, 0, 12+nudge>, < 1.2, 0, 12+nudge>, 1 }
    + object { cylinder < 0,-1.2, 13+nudge>, < 0, 1.2, 13+nudge>, 1 }
    + object { cylinder <-1.2, 0, 14+nudge>, < 1.2, 0, 14+nudge>, 1 }
    + object { cylinder < 0,-1.2, 15+nudge>, < 0, 1.2, 15+nudge>, 1 }
    + object { cylinder <-1.2, 0, 16+nudge>, < 1.2, 0, 16+nudge>, 1 }
    & ~object {cylinder < 0, 0,-20>, < 0, 0, 20>, 1}
    texture {surface {color coral ambient 0.3 diffuse 0.8}}
}
launch_tube
// chaser lights
define trl1 -20 + fmod(frame, 16) + nudge
define trl2 -4 + fmod(frame, 16) + nudge
define pi 3.14159
spot_light white,<10,0,trl1-5>,<0,0,trl1>,3,5,18
spot_light white,<0,10,trl1-5>,<0,0,trl1>,3,5,18
spot_light white,<-10,0,trl1-5>,<0,0,trl1>,3,5,18
spot_light white,<0,-10,trl1-5>,<0,0,trl1>,3,5,18
spot_light white,<10,0,trl2-5>,<0,0,trl2>,3,5,18
spot_light white,<0,10,trl2-5>,<0,0,trl2>,3,5,18
spot_light white,<-10,0,trl2-5>,<0,0,trl2>,3,5,18
spot_light white,<0,-10,trl2-5>,<0,0,trl2>,3,5,18
```

How It Works

The fun begins with the structured definition of the rocket. It is constructed in stages from cylinders one unit long (minus the slot) centered on the origin, 0.2 units in diameter, that we call *section*. The variable *rocketz* establishes the forward motion of the rocket along the z axis under the control of the frame counter. We define nine points based on *rocketz* strung out in a line, t1 - t9, which specify the offsets for each section of the rocket. We make nine copies of the original *section* plus a nose cone, paint it red and call it *rocket* (Figure 2-11). The engine is a bluish light inside a sphere traveling along inside the

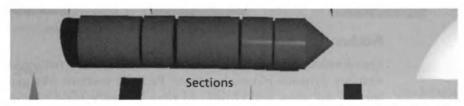


Figure 2-11 Our rocket

middle of the last stage. The shadow flags have been turned off for this sphere to let the light pass through. There's more stuff about shadow flags in Section 3.3.

The rocket originally had nine lights inside it, but all save one were commented out when the chaser lights were added. We've already covered how *nudge* moves the tunnel a bit forward each frame and accomplishes the treadmill motion.

The traveling lights were done with four external spots outside the tunnel. The tunnel was made airy enough to allow them to enter. Moving lights inside the tunnel didn't produce quite the same effect. They shine everywhere and don't produce the kind of localized pools of light that are needed to represent tight areas of force.

Comments

The tunnel was stepped to repeat every 20 frames, while the lights repeated every 16 frames. This was arbitrary, but it had repercussions. For a smooth loop point, the frame count now must be some multiple of the lowest common denominator of 16 and 20. Frame counts of 80 or 160 are the two lowest ones. Changing

define nudge 2 - fmod(frame,20) / 10

to

define nudge 2 - fmod(frame, 16) / 8

allows it to loop smoothly every 16 frames, giving five times more potential loop points. Provided the animation is some multiple of 16 frames long, and the rocket has passed far enough away that its sudden disappearance isn't too noticeable, the animation will loop smoothly.



2.3 How do I...

Use functions to control the positions of objects?

You'll find the code for this in: PLY\CHAPTER2\BOLITA

Problem

Coordination of several objects in motion is frequently required to accurately simulate dynamic physical systems. Periodic motions like pendulums and springs use sines and cosines to time-sequence their positions, and for multibody collections of objects, there needs to be some coordination between

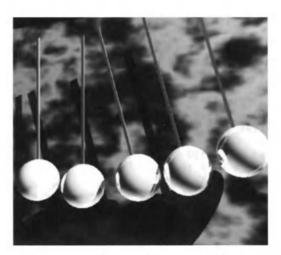


Figure 2-12 Simplified view of a popular 60's office ornament

the movements of the various parts. Take for example a series of pendulums that interact with one another, a popular office ornament and common to high school physics labs in the 60s and 70s. It was simply a line of metal balls attached to strings that would swing back and forth and bang into each other. Its appeal was that it demonstrated conservation of momentum. A ball striking one end would set the ball at the other end in motion.

David Mason posted an animation called "DESKTOY" on the "You Can Call Me Ray" BBS (YCCMR) back in September of '91 illustrating this motion, Jorge Arrequin posted a similar animation called "BOLITA," and quite independently Hank Palczewski posted yet another example called "KLICKL" on CompuServe in March '93. Hank's modeling was the most elaborate, but all these animations display basically the same thing. A simple representation with three balls swinging is shown in Figure 2-12.

Technique

We'll create the basic simulation in QuickBasic, translate that into Polyray, and then proceed to embellish the basic motion with additional periodic motions.

Steps

We can generate a wire frame representation of this with the QuickBasic code in this BOLITA animation.

```
' BOLITA - Based on Jorge Arreguins Bolita Animation
' motion - a linear group of balls swings back and forth
' the number of moving balls is set with the variable moving
DECLARE SUB rotate (x, y, z)
SCREEN 12
WINDOW (-10, -2)-(10, 11)
LINE (-10, -1)-(10, 10), B
pi = 3.14159
rad = pi / 180
moving = 3
                         ' number of balls in motion at any time
DO WHILE INKEY$ = ""
                         ' go until a key is pressed
t = t + 6
                         ' time step size
FOR b = 1 TO 5
                         ' lets do 5 balls, rocking
phz = 30 * SIN(t * rad) ' 30 about
                         ' 270 (6 oclock)
ang = 270 + phz
' =========CALCULATE============
' 5 balls placed at x = -4, -2, 0, +2, +4 (b*2-6)
' during the positive half of the swing (phz>0)
' displace those balls on the right where b
' (the ball count) is greater than (5-moving)
' i.e., moving = 3, 5-moving =2, where b is greater
' than 3, 4 and 5. Otherwise, maintain the balls
' at their rest position.
IF phz > 0 AND b > (5 - moving) THEN
  x(b) = 8 * COS(ang * rad) + b * 2 - 6
  y(b) = 8 * SIN(ang * rad) + 8
  x(b) = b * 2 - 6
  y(b) = 0
END IF
' =======OTHER HALF OF SWING==========
' during the negative half of the swing,
' move only those balls where b is less than
' moving +1
' e.i., moving = 3, moving+1 = 4, move the balls
' 1, 2 and 3. Balls 4 and 5 will have swung to their
' rest position and will then just appear to stop.
IF phz < 0 AND b < moving + 1 THEN
  x(b) = 8 * COS(ang * rad) + b * 2 - 6
  y(b) = 8 * SIN(ang * rad) + 8
END IF
```

```
NEXT b
' DISPLAY

FOR b = 1 T0 5
    'undraw the old screen (draw in black)
    LINE (b * 2 - 6, 8)-(ox(b), oy(b)), 0
    CIRCLE (ox(b), oy(b)), 1, 0

    'draw the new screen
    LINE (b * 2 - 6, 8)-(x(b), y(b)), 15
    CIRCLE (x(b), y(b)), 1, 15

    'save
    ox(b) = x(b)
    oy(b) = y(b)

NEXT b
Loop
```

Features Deserve to be Played With

By including a variable *moving* in the preceding program, a hook placed in the code allows easy change in the number of balls in motion from 0 to 5 simply by changing one variable. If a button is there, we must press it. We'll change the number of balls swinging during this animation. We'll use the static variable *moving* which will be bumped up by 1 every time the balls complete one cycle (move forward by 360°). A frame produced by the following code (BOLITA.PI) is shown in Figure 2-13.

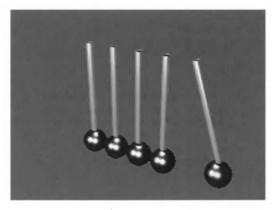


Figure 2-13 Output of BOLITA1.Pl

```
// BOLITA1.PI
// Bumping Line o' Balls
start_frame 0
end_frame 179
total_frames 180
outfile balls
include "\PLY\COLORS.INC"
// set up background color & lights
background Navyblue
light <10, 10, -20>
light <-10, 10, -20>
light <0, 20, 0>
// set up the camera
viewpoint {
   from <11, 15,-20>
   аt
       <0, 4, 0>
  up <0, 1, 0>
   angle 30
   resolution 320,200
   aspect 1.43
   }
define pi 3.14159
define rad pi / 180
if (frame==start_frame) {
   static define t O
  static define moving 1
define phz 30 * sin(t * rad)
define ang 270 + phz
// ball 1
   if (phz > 0 && 1 > (5 - moving)) {
      define x1 8 * cos(ang * rad) + 1 * 2 - 6
     define y1 8 * sin(ang * rad) + 8
  }
  else {
     define x1 1 * 2 - 6
     define y1 0
  }
  if (phz < 0 && 1 < (moving + 1)) {
     define x1 8 * cos(ang * rad) + 1 * 2 - 6
```

```
define y1 8 * sin(ang * rad) + 8
   }
// ball 2
   if (phz > 0 && 2 > (5 - moving)) {
      define x2 8 * cos(ang * rad) + 2 * 2 - 6
      define y2 8 * sin(ang * rad) + 8
   else {
      define x2 2 * 2 - 6
      define y2 0
   if (phz < 0 \&\& 2 < (moving + 1)) {
      define x2 \ 8 \ * \cos(ang \ * \ rad) + 2 \ * 2 - 6
      define y2 8 * sin(ang * rad) + 8
   }
// ball 3
   if (phz > 0 && 3 > (5 - moving)) {
      define x3 \ 8 \ * \cos(ang \ * \ rad) \ + \ 3 \ * \ 2 \ - \ 6
      define y3 8 * sin(ang * rad) + 8
   }
   else {
      define x3 3 * 2 - 6
      define y3 0
   }
   if (phz < 0 && 3 < (moving + 1)) {
      define x3 \ 8 \ * \cos(ang \ * \ rad) \ + \ 3 \ * \ 2 \ - \ 6
      define y3 8 * sin(ang * rad) + 8
   }
// ball 4
   if (phz > 0 && 4 > (5 - moving)) {
      define x4 8 * cos(ang * rad) + 4 * 2 - 6
      define y4 8 * sin(ang * rad) + 8
   }
   else {
      define x4 4 * 2 - 6
      define y4 0
   }
   if (phz < 0 && 4 < (moving + 1)) {
      define x4 \ 8 \ * \cos(ang \ * \ rad) + 4 \ * 2 - 6
      define y4 8 * sin(ang * rad) + 8
   }
// ball 5
```

```
continued from previous page
   if (phz > 0 \&\& 5 > (5 - moving)) {
      define x5.8 * cos(ang * rad) + 5 * 2 - 6
      define y5 8 * sin(ang * rad) + 8
   }
   else {
     define x55 * 2 - 6
      define y5 0
   }
   if (phz < 0 && 5 < (moving + 1)) {
      define x5.8 * cos(ang * rad) + 5 * 2 - 6
      define y5 8 * sin(ang * rad) + 8
   }
   define z 0
   object { sphere <x1,y1,z>,1 shiny_blue }
   object { sphere <x2,y2,z>,1 shiny_blue }
   object { sphere <x3,y3,z>,1 shiny_blue }
   object { sphere <x4,y4,z>,1 shiny_blue }
   object { sphere <x5,y5,z>,1 shiny_blue }
   object { cylinder <x1,y1+1,z>,<1*2-6,8,0>,0.25 shiny_coral }
   object { cylinder <x2,y2+1,z>,<2*2-6,8,0>,0.25 shiny_coral }
   object { cylinder <x3,y3+1,z>,<3*2-6,8,0>,0.25 shiny_coral }
   object { cylinder <x4,y4+1,z>,<4*2-6,8,0>,0.25 shiny_coral }
   object { cylinder <x5,y5+1,z>,<5*2-6,8,0>,0.25 shiny_coral }
   static define t t+10
   if (fmod(t,360)==0) static define moving moving+1
```

How It Works

Each ball is controlled by its own *if-else* block. We "unrolled" the loop in the QuickBasic code and hard-coded the decision values one at a time. For example, the code controlling ball 5 is:

```
if (phz > 0 && 5 > (5 - moving))
```

The variable t adds another swinging ball every 36 frames. For a 180 frame animation, we start with one ball swinging, and end up with five balls swinging. Since the balls will all be at rest at the beginning and end of the animation, it loops nicely, although the number of balls swinging goes 1-2-3-4-5-1-2-3-4-5... The jump between five and one balls moving is disconcerting. It would feel better if the number of balls moving changed by only one each cycle. Using the @list feature in DTA, we can force the animation to loop 1-2-3-4-5-4-3-2-1-2-3-4-5... by creating a list file that uses frames 0-179, then 108-143, 72-107, and 36-71. Otherwise, the code is

identical to the QuickBasic version. Single cylinders are used as our string to attach the balls to an invisible frame.

Improving the Model

We've got the basic motion. Now we need to build a better frame to hold the balls. The rods extending up out of the balls are too thick, they need to be more like threads, and there needs to be two of them attached to parallel supports above the balls to keep them centered. We'll end up with your basic V shape.

Flapping Wings

For grins, let's make the support frame V flap its wings like a bird. A flap can be created as an asymmetrical wave form, with a slow smooth upward motion followed by a rapid downward motion, as illustrated in the wave form in Figure 2-14.

This is generated by periodically adding a hump (a Gaussian function) to a sine wave. The height of the balls attached to the wings should follow this periodic motion, but in the opposite direction. When the wings are moving down and spreading apart, the balls should be moving up. The following program (FLAP.BAS) draws the wave form at the bottom of Figure 2-14, pauses, then flaps the frame at the top of the figure.

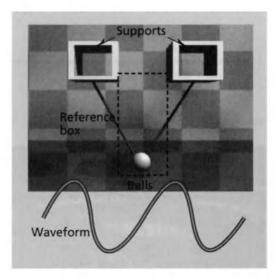


Figure 2-14 Flapping supports and the wave form they follow. The box was added as a spatial reference

```
' FLAP.BAS
SCREEN 12
WINDOW (-16, -12)-(16, 12)
pi = 3.14159
rad = pi / 180
frame = -16
'show the waveform
FOR angle = -360 TO 360 STEP 1
       x = angle / 30
       w = 10 * COS(angle * rad)
       z = EXP(1.6 * (1 + COS((angle - 80) * rad)))
       IF angle = -360 THEN PSET (x, y / 5) ELSE LINE -(x, y / 5)
NEXT angle
DO WHILE INKEY$ = "": LOOP
CLS
' a reference box
LINE (-4, 0)-(4, 8), B
DO WHILE INKEY$ = "" go until a key is pressed
 t = t + 10
                          ' time step size
' ===========The Assymetric Waveform============
       w = 10 * COS(t * rad)
                                            ' periodic motion
       z = EXP(1.6 * (1 + COS((t - 80) * rad))) ' periodic hump
       phz = w + z
                                               ' add 'em together
                                              ' shift the phase
       ang = 45 + phz
x = 8 * COS(ang * rad)
       y = 8 * SIN(ang * rad) + 2
       fx = 0
       fy = -4 * SIN(ang * rad) + 4
   ' undraw
       LINE (ofx, ofy)-(ox, oy), 0
       LINE (ofx, ofy)-(-ox, oy), 0
       CIRCLE (ox, oy), 1, 0
       CIRCLE (-ox, oy), 1, 0
       CIRCLE (ofx, ofy -.5), .5, 0
   ' draw
       LINE (fx, fy)-(x, y), 15 ' draw the current screen
       LINE (fx, fy)-(-x, y), 15 ' draw the current screen
      CIRCLE (x, y), 1, 15 ' (draw in color 15 = white)
CIRCLE (-x, y), 1, 15 ' (draw in color 15 = white)
       CIRCLE (fx, fy - .5), .5, 15
```

First Example—Running in Place

Now we'll place this flapping, swinging-ball thing above another periodic surface—a rippling pool of water (as seen in Figure 2-15). The script for this animation (BOLITA2.PI) is shown in the following listing.

```
// BOLITA2.PI
// Bumping, Flapping Line o' Balls over a Pool
start_frame 0
end_frame 179
total_frames 180
define index frame/total_frames
outfile balls
include "\PLY\COLORS.INC"
// set up background color & lights
background Navyblue
light <10, 10, -20>
light <-10, 10, -20>
light <0, 20, 0>
// set up the camera
```

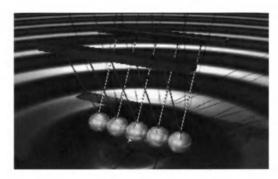


Figure 2-15 Bolitas over water

CHAPTER TWO

```
continued from previous page
viewpoint {
   from <12, 16,-22>
   at
      <0.5, 5, 0>
   up
        <0, 1, 0>
   angle 30
   resolution 320,200
   aspect 1.43
define pi 3.14159
define rad pi / 180
// may need to adjust based on where you start
if (frame==start_frame) {
   static define moving O
}
define t frame*10
define phz 30 * sin(t * rad)
define ang 270 + phz
// ball 1
   if (phz > 0 && 1 > (5 - moving)) {
      define x1 8 * cos(ang * rad) + 1 * 2 - 6
      define y1 8 * sin(ang * rad) + 8
   }
   else {
      define x1 1 * 2 - 6
      define y1 0
   }
   if (phz < 0 && 1 < (moving + 1)) {
      define x1 8 * cos(ang * rad) + 1 * 2 - 6
      define y1 8 * sin(ang * rad) + 8
   }
// ball 2
   if (phz > 0 && 2 > (5 - moving)) {
      define x2 \ 8 \ * \cos(ang \ * \ rad) + 2 \ * 2 - 6
      define y2 8 * sin(ang * rad) + 8
   }
   else {
     define x2 2 * 2 - 6
     define y2 0
   if (phz < 0 && 2 < (moving + 1)) {
     define x2 \ 8 * cos(ang * rad) + 2 * 2 - 6
     define y2 8 * sin(ang * rad) + 8
```

```
}
// ball 3
   if (phz > 0 && 3 > (5 - moving)) {
      define x3 \ 8 \ * \cos(ang \ * \ rad) + 3 \ * 2 - 6
      define y3 8 * sin(ang * rad) + 8
   }
   else {
      define x3 \ 3 \ * \ 2 \ - \ 6
      define y3 0
   }
   if (phz < 0 && 3 < (moving + 1)) {
      define x3 \ 8 \ * \cos(ang \ * \ rad) + 3 \ * 2 - 6
      define y3 8 * sin(ang * rad) + 8
   }
// ball 4
   if (phz > 0 && 4 > (5 - moving)) {
      define x4 8 * cos(ang * rad) + 4 * 2 - 6
      define y4 8 * sin(ang * rad) + 8
   }
   else {
      define x4 4 * 2 - 6
      define y4 0
   }
   if (phz < 0 && 4 < (moving + 1)) {
      define x4 8 * cos(ang * rad) + 4 * 2 - 6
      define y4 8 * sin(ang * rad) + 8
   }
// ball 5
   if (phz > 0 && 5 > (5 - moving)) {
      define x5 \ 8 \ * \cos(ang \ * \ rad) \ + \ 5 \ * \ 2 \ - \ 6
      define y5 8 * sin(ang * rad) + 8
   7
   else {
      define x5 5 * 2 - 6
      define y5 0
   if (phz < 0 \&\& 5 < (moving + 1)) {
      define x5 \ 8 \ * \cos(ang \ * rad) + 5 \ * 2 - 6
      define y5 8 * sin(ang * rad) + 8
   }
// =======make the mounts flap like a bird============
define phz2 10 * \cos(2*t * rad) + \exp(1.6 * (1 + \cos((2*t - 80) * rad)))
                                                                     continued on next page
```

```
continued from previous page
define ang 45 + phz^2
define zm 8 * COS(ang2 * rad)
define ym 8 * SIN(ang2 * rad) + 2
// this term moves the balls slightly up 180 out of phase with the
// flapping of the support
define fy -4 * SIN(ang2 * rad) + 4
define z O
define slateish
texture {
   surface {
      ambient SlateBlue, 0.2
      diffuse SlateBlue, 0.6
      specular white, 0.6
      reflection white, 0.5
      microfacet Reitz 10
   }
object { sphere <x1,y1+fy,z>,1 slateish }
object { sphere <x2,y2+fy,z>,1 slateish }
object { sphere <x3,y3+fy,z>,1 slateish }
object { sphere <x4,y4+fy,z>,1 slateish }
object { sphere <x5,y5+fy,z>,1 slateish }
// The support threads, one on either side of the line of balls
// running up to the supports
object { cylinder \langle x1,y1+1+fy,z\rangle,\langle 1*2-6,ym,-zm\rangle,0.05 shiny_orange }
object { cylinder \langle x^2, y^2+1+fy, z\rangle, \langle z^2+2-6, ym, -zm\rangle, 0.05 shiny_orange }
object { cylinder \langle x3,y3+1+fy,z\rangle,\langle 3*2-6,ym,-zm\rangle,0.05 shiny_orange }
object { cylinder \langle x4,y4+1+fy,z\rangle,\langle 4*2-6,ym,-zm\rangle,0.05 shiny_orange }
object { cylinder <x5,y5+1+fy,z>,<5*2-6,ym,-zm>,0.05 shiny_orange }
object { cylinder <x1,y1+1+fy,z>,<1*2-6,ym,zm>,0.05 shiny_orange }
object { cylinder \langle x^2, y^2+1+fy, z\rangle, \langle 2*2-6, ym, zm\rangle, 0.05 shiny_orange }
object { cylinder \langle x3,y3+1+fy,z\rangle,\langle 3*2-6,ym,zm\rangle,0.05 shiny_orange }
object { cylinder <x4,y4+1+fy,z>,<4*2-6,ym,zm>,0.05 shiny_orange }
object { cylinder <x5,y5+1+fy,z>,<5*2-6,ym,zm>,0.05 shiny_orange }
define real_dark <0.1,0.1,0.1>
define onyx texture { metallic { color real_dark } }
// two rectangular support structures for the line of balls
object { box <-6,ym-0.1,-zm-0.5>,<6,ym+0.5,-zm+0.5> onyx }
object { box <-6,ym-0.1, zm-0.5>, <6,ym+0.5, zm+0.5> onyx }
define blue_ripple
```

```
texture {
   noise surface {
      color navyblue
      normal 2
      frequency 0.5
      phase -2*pi*t/360
      bump scale 2
      ambient 0.3
      diffuse 0.4
      specular yellow, 0.7
      reflection 0.5
      microfacet Reitz 10
   }
// make a watery floor
object {
   polygon 4,<-100,-1.2,-8>,<24,-1.2,-8>,<24,-1.2,800>,<-100,-1.2,800>
   blue_ripple
}
if (fmod(t,360)==0) static define moving moving + 1
```

The variable *phz2* controls the flapping of the two support structures, which control *ym* and *zm* which in turn are used to make two long rectangular boxes. We attach threads running from these supports to the swinging balls. A *blue_ripple* texture is defined which through a simple trick of surface normals, makes a perfectly flat four-sided polygon appear to become a rippling fluid surface, moving in sync with our Bolita structure.

Second Example—Flying About

The previous animation has our creation hovering above one spot. It really ought to go somewhere. Let's make it fly down a tunnel with a shiny floor. We'll make a seemingly endless stream of them fly past the camera. This is basically a 36-frame animation. If we take three copies of Bolita, space them 36 units apart, and move them all forward one unit per frame, the copies will line up every 36 frames. This can be managed as follows:

```
define copies fmod(frame,36)
bolita { translate <-66+copies,0,0> }
bolita { translate <-30+copies,0,0> }
bolita { translate < 6+copies,0,0> }
```

All we need to do now is hide the places where the copies suddenly appear and disappear, and the illusion of an endless procession is complete (Figure 2-16). The next listing, provides the code for this iteration, which is called BOLLITA3.PL

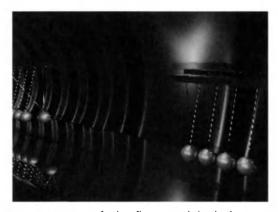


Figure 2-16 Line of Bolitas flying toward Cleveland

```
// BOLITA3.PI
// Bumping Line o' Balls, Infinite Procession, Flapping down a Tunnel
start_frame 0
end_frame 179
total_frames 180
outfile balls
include "PLY\COLORS.INC"
// set up background color & lights
background Navyblue
define dim <0.7,0.7,0.7>
light dim,<10, 10, -20>
light dim, <-10, 10, -20>
light <0, 20, 0>
// set up the camera
viewpoint {
   from <12, 12,-22>
        <0.5, 5, 0>
   аt
      <0, 1, 0>
   up
   angle 50
   resolution 320,200
   aspect 1.43
   }
define pi 3.14159
define rad pi / 180
define speed 10
define t frame*speed
```

```
define moving floor (frame/(360/speed)) + 1
define phz 30 * sin(t * rad)
define ang 270 + phz
// ball 1
   if (phz > 0 && 1 > (5 - moving)) {
      define x1 8 * cos(ang * rad) + 1 * 2 - 6
      define y1 8 * sin(ang * rad) + 8
   }
   else {
      define x1 \ 1 \ * \ 2 \ - \ 6
      define y1 0
   if (phz < 0 && 1 < (moving + 1)) {
      define x1 8 * cos(ang * rad) + 1 * 2 - 6
      define y1 8 * sin(ang * rad) + 8
   }
// ball 2
   if (phz > 0 && 2 > (5 - moving)) {
      define x2 \ 8 \ * \cos(ang \ * \ rad) + 2 \ * 2 - 6
      define y2 8 * sin(ang * rad) + 8
   }
   else {
      define x2 2 * 2 - 6
      define y2 0
   }
   if (phz < 0 && 2 < (moving + 1)) {
      define x2 \ 8 \ * \cos(ang \ * \ rad) \ + \ 2 \ * \ 2 \ - \ 6
      define y2 8 * sin(ang * rad) + 8
   }
// ball 3
   if (phz > 0 && 3 > (5 - moving)) {
      define x3 \ 8 \ * \cos(ang \ * \ rad) \ + \ 3 \ * \ 2 \ - \ 6
      define y3 8 * sin(ang * rad) + 8
   7
   else {
      define x3 3 * 2 - 6
      define y3 0
   if (phz < 0 && 3 < (moving + 1)) {
      define x3 \ 8 \ * \cos(ang \ * \ rad) \ + \ 3 \ * \ 2 \ - \ 6
      define y3.8 * sin(ang * rad) + 8
   }
```

```
continued from previous page
// ball 4
   if (phz > 0 && 4 > (5 - moving)) {
      define x4 8 * cos(ang * rad) + 4 * 2 - 6
      define y4 8 * sin(ang * rad) + 8
   else {
      define x4 4 * 2 - 6
      define y4 0
   }
   if (phz < 0 && 4 < (moving + 1)) {
      define x4 8 * cos(ang * rad) + 4 * 2 - 6
      define y4 8 * sin(ang * rad) + 8
   }
// ball 5
   if (phz > 0 && 5 > (5 - moving)) {
      define x5 \ 8 \ * \cos(ang \ * \ rad) \ + \ 5 \ * \ 2 \ - \ 6
      define y5 8 * sin(ang * rad) + 8
   }
   else {
      define x55 \times 2 - 6
      define y5 0
   }
   if (phz < 0 && 5 < (moving + 1)) {
      define x5 \ 8 \ * \cos(ang \ * \ rad) \ + \ 5 \ * \ 2 \ - \ 6
      define y5 8 * sin(ang * rad) + 8
   }
// make the mounts flap like a bird
   define phz2 10 * \cos(2*t * rad) + \exp(1.6 * (1 + \cos((2*t - 80) * rad)))
   define ang2 45 + phz2
   define zm 8 * COS(ang2 * rad)
   define ym 8 * SIN(ang2 * rad) + 2
   define fy -4 * SIN(ang2 * rad) + 4
   define z O
   define slateish
   texture {
      surface {
         ambient SlateBlue, 0.2
         diffuse SlateBlue, 0.6
         specular white, 0.6
         reflection white, 0.5
         microfacet Reitz 10
         }
```

```
}
   define real_dark <0.1,0.1,0.1>
   define onyx texture { metallic { color real_dark } }
   define blue_ripple
   texture {
      noise surface {
         color navyblue
         normal 2
         frequency 1
         phase -2*pi*t/360
         bump scale 2
         ambient 0.2
         diffuse 0.4
         specular yellow, 0.5
         reflection 0.25
         microfacet Reitz 10
         }
      }
   define light_blue_ripple
   texture {
      noise surface {
         color navyblue
         normal 1
         frequency 1
         phase -2*pi*t/360
         bump_scale 0.2
         ambient 0.2
         diffuse 0.4
         specular yellow, 0.5
         reflection 0.25
         microfacet Reitz 10
         }
      }
// the flapping clacking contraption
   define bolita
      object {
         object { sphere <x1,y1+fy,z>,1 slateish }
       + object { sphere <x2,y2+fy,z>,1 slateish }
       + object { sphere <x3,y3+fy,z>,1 slateish }
       + object { sphere <x4,y4+fy,z>,1 slateish }
       + object { sphere <x5,y5+fy,z>,1 slateish }
   // the strings
       + object {cylinder <x1,y1+1+fy,z>,<1*2-6,ym,-zm>,0.05 shiny_orange }
       + object (cylinder \langle x^2, y^2+1+fy, z\rangle, \langle z^*2-6, ym, -zm\rangle, 0.05 shiny_orange )
       + object {cylinder <x3,y3+1+fy,z>,<3*2-6,ym,-zm>,0.05 shiny_orange }
                                                                    continued on next page
```

continued from previous page + object {cylinder $<x4,y4+1+fy,z>,<4*2-6,ym,-zm>,0.05 shiny_orange}}$ + object {cylinder <x5,y5+1+fy,z>,<5*2-6,ym,-zm>,0.05 shiny_orange } + object { cylinder $\langle x1,y1+1+fy,z\rangle,\langle 1*2-6,ym,zm\rangle,0.05$ shiny orange } + object { cylinder <x2,y2+1+fy,z>,<2*2-6,ym,zm>,0.05 shiny_orange } + object { cylinder <x3,y3+1+fy,z>,<3*2-6,ym,zm>,0.05 shiny_orange } + object { cylinder $\langle x4,y4+1+fy,z\rangle,\langle 4*2-6,ym,zm\rangle,0.05$ shiny_orange } + object { cylinder $\langle x5,y5+1+fy,z\rangle,\langle 5*2-6,ym,zm\rangle,0.05$ shiny_orange } // the mounts + object { box <-6,ym-0.1,-zm-0.5>,<6,ym+0.5,-zm+0.5> onyx } + object { box $<-6,ym-0.1, zm-0.5>, <6,ym+0.5, zm+0.5> onyx }$ } define copies fmod(frame, 36) bolita { translate <-66+copies,0,0> } bolita { translate <-30+copies,0,0> } bolita { translate < 6+copies,0,0> } // make watery walls object { object {cylinder <-100,0,0>,<100,0,0>,16} & object {box <-101,-8,-20>,<101,20,20>} rotate <125,0,0> translate <0,4,5> blue_ripple } // reflecting pool object { polygon 4,<-100,-4,-10>,<-100,-4,10>,<100,-4,10>,<100,-4,-10> }

A watery tunnel with a shiny floor was created by mapping our *blue_ripple* texture on a cylinder. Cutting the side out of it allows the camera to see inside. The shiny floor was added to clean up the edge of the cylinder and double the apparent number of objects moving in the scene.

Comments

It's possible to continue adding more and more periodic motions all linked to the basic motions we've created. Rather than a constant forward linear motion, we could make this motion surge and glide. We could make the entire frame swoop up and down, simulating the real motions of birds as they flap their wings. We could periodically halt the flapping or change its rate. We just keep nesting the motions.



Create a five-dimensional dripping faucet?

You'll find the code for this in: PLY\CHAPTER2\DRIP

Problem

Creating convincing animations of fluids in motion cannot be done with conventional geometric primitives like spheres and cylinders. Polyray supports a variety of tear-drop shaped objects whose outer surfaces are defined by some polynomial equation (check out PIRIFORM.PI and TEAR5.PI in the POLY directory), and it might be possible through careful selection of coefficients to produce a dripping faucet animation with a couple of them. However, an easier way to do this is to use a primitive known as a blob, which acts much as its name suggests. It's so useful for generating really interesting animations that we're devoting an entire chapter to it later on (Chapter 7). As an introduction, our simple dripping faucet animation shows off what a blob is capable of and how to use it.

Technique

Try to humor me on this one, OK? A dripping faucet in three dimensions has a drop moving down a single axis. By extension, a 4-D dripping faucet would drip simultaneously down two perpendicular axes, and a 5-D faucet would drip down three perpendicular axes. If we can imagine three orthogonal gravity fields, let's mirror them, and cause our drops to go in six different directions at the same time, namely down both positive and negative Cartesian axes. Droplets form out of a central transparent ball and exit in six different directions simultaneously.

This animation exploits the fact that metaballs (the things blobs are made of) can hide inside each other and that when two metaballs move apart, they can form little football shapes and then disappear altogether. Figure 2-17 shows a series of blobs comprised of two metaballs. The metaballs have a maximum interaction distance set at 1.0. At this distance of 1.0, they're footballing, and by 1.1, they're gone.

Steps

5DRIP.PI starts with three metaballs per axis, for a total of 18, spaced one unit apart (Figure 2-18), and makes them all move away from the origin by multiplying their distance from the origin by a linearly increasing factor

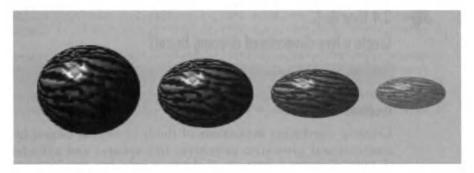


Figure 2-17 As the two metaballs—which define a blob—move farther apart, the blob forms a football shape, then vanishes

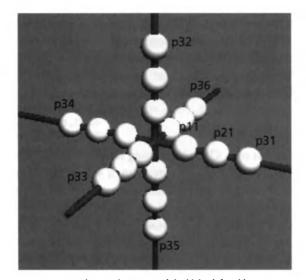


Figure 2-18 The initial positions of the blobs defined by p11-p36

called *framer* (the frame counter divided by 25). They not only move away from the origin, but they also move away from each other. As they move farther and farther apart, they make the blob become longer and narrower and finally disappear. We'll use transparent blobs over a checkerboard so that the refraction will emphasize the surface deformation of the evolving blob.

```
// 5DRIP.PI 5D Dripping Faucet Animation
//
// Polyray input file - Jeff Bowermaster
// define the range of the animation
start_frame 0
```

```
end_frame
             90
total_frames 91
outfile drip
// set up the camera
viewpoint {
   from <8,8,-8>
   at <0,0,0>
   up <0,1,0>
   angle 45
   resolution 320,240
   aspect 1.333
   }
include "\PLY\COLORS.INC"
// set up background color & lights
background skyblue
light <-10,20,-50>
light < 10,20,-50>
// create a ground plane
object {
   polynomial y + 0.01
   texture { checker matte_white, matte_black }
   scale <5, 5, 5>
   translate <0,-10,0>
define blue_glass
texture {
   surface {
      ambient blue, 0.2
      diffuse blue, 0.6
      specular 0.6
      reflection white, 0.1
      transmission white, 0.95, 1.5
      }
   }
// give the initial locations of the blob points
define p11
             < 1, 0, 0 >
             < 0, 1, 0 >
define p12
define p13
             < 0, 0, 1 >
define p14
             <-1, 0, 0 >
define p15
             < 0,-1, 0 >
define p16
             < 0, 0,-1 >
define p21
             < 2, 0, 0 >
define p22
             < 0, 2, 0 >
define p23
             < 0, 0, 2 >
```

```
continued from previous page
define p24
           <-2, 0, 0 >
define p25 < 0,-2, 0 >
define p26 < 0, 0, -2 >
define p31
             < 3, 0, 0 >
             < 0, 3, 0 >
define p32
define p33
           < 0, 0, 3 >
define p34 <-3, 0, 0 >
define p35 < 0,-3, 0 >
define p36
             < 0, 0,-3 >
// a blob explodes along the xyz axes
define framer frame/25
object {
   blob 0.10:
      0.1, 3.0, p11*framer,
      0.1, 3.0, p12*framer,
      0.1, 3.0, p13*framer,
      0.1, 3.0, p14*framer,
      0.1, 3.0, p15*framer,
      0.1, 3.0, p16*framer,
      0.1, 3.0, p21*framer,
      0.1, 3.0, p22*framer,
      0.1, 3.0, p23*framer,
      0.1, 3.0, p24*framer,
      0.1, 3.0, p25*framer,
      0.1, 3.0, p26*framer,
      0.1, 3.0, p31*framer,
      0.1, 3.0, p32*framer,
      0.1, 3.0, p33*framer,
      0.1, 3.0, p34*framer,
      0.1, 3.0, p35*framer,
      0.1, 3.0, p36*framer,
      0.5, 3.0, <0,0,0> // Master Blob
   root_solver Ferrari
   u_steps 20
  v_steps 20
   blue_glass
```

How It Works

After defining a viewpoint and lights, we create a checkered background plane as follows:

```
// Create a ground plane
object {
  polynomial y + 0.01
  texture { checker matte_white, matte_black }
```

```
scale <5, 5, 5>
translate <0,-10,0>
```

This step uses a first order (no powers above 1) polynomial syntax. The plane is moved slightly above the zero point (+0.01) to avoid digital zits, a problem caused by roundoff errors when applying a periodic texture (like a checkerboard) exactly at the point where it changes sign, in this case from black to white. We define 18 vectors (*p11-p36*) which describe the positions of the metaballs as three points one unit apart along the three positive and three negative Cartesian axes (Figure 2-18).

These are multiplied by *framer* in the blob object definition to move these control elements away from the origin at a constant velocity. The last metaball in the blob is a large fixed sphere at the origin large enough to hold all the other metaballs at the beginning of the animation. The outer metaballs move at three times the velocity of the inner ones, and after 90 frames, all the metaballs are so far apart that the only one left is the fixed sphere.

Embellishments

A more interesting background can be achieved by creating two marbly textures that are the exact inverses of each other. We'll call them <code>annoying_marble1</code> and <code>annoying_marble2</code>, because rapidly switching between them annoys either the monitor (which flickers) or your eyes (which water). The next iteration moves this texture forward across the checkerboard and shifts the checkerboard forward during the animation as well. As the marble patterns move across the checker borders, it goes from negative to positive, but your eyes see it as a continuation of the same pattern.

```
//
// 5DRIP1.PI 5D Dripping Faucet Animation
// (formerly 4D or 3DRIP... inflation<g>)
// Polyray input file - Jeff Bowermaster

// define the range of the animation
start_frame 0
end_frame 89
total_frames 90

outfile drip1
define shift frame/30

// set up the camera
viewpoint {
  from <8,8,-8>
  at <0,0,0>
```

CHAPTER TWO

```
continued from previous page
   up <0,1,0>
   angle 45
   resolution 320,200
   aspect 1.43
   }
// set up background color & lights
background skyblue
spot_light white,< 0,50,0>,<0,0,0>, 3, 15, 30
include "\ply\colors.inc"
define blue_glass
texture {
   surface {
                                    // when experimenting, save old values
      ambient yellow, 0.1
                                    // 0.2
                                    // 0.6
      diffuse midnightblue, 0.2
                                    // 0.6
      specular 0.9
      reflection white, 0.05
                                    // 0.1
      microfacet Phong 2
      transmission white, 0.98, 1.5 // 0.95
      }
   }
// give the initial locations of the blob points
define p11
            < 1, 0, 0 >
define p12
            < 0, 1, 0 >
           < 0, 0, 1 >
define p13
define p14 <-1, 0, 0 >
define p15 < 0,-1, 0 >
define p16
            < 0, 0,-1 >
define p21
            < 2, 0, 0 >
           < 0, 2, 0 >
define p22
define p23 < 0, 0, 2 >
define p24 <-2, 0, 0 >
define p25
            < 0,-2, 0 >
            < 0, 0,-2 >
define p26
            < 3, 0, 0 >
define p31
define p32 < 0, 3, 0 >
            < 0, 0, 3 >
define p33
define p34
           <-3, 0, 0 >
            < 0,-3, 0 >
define p35
            < 0, 0,-3 >
define p36
define annoying_marble1
texture {
   noise surface {
      color white
```

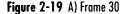
```
position_fn 1
      lookup_fn 1
      octaves 3
      turbulence 3
      ambient 0.3
      diffuse 0.8
      specular 0.3
      microfacet Reitz 5
      color_map(
         [0.0, 0.8, Yellow, Yellow*0.6]
         [0.8, 1.0, Yellow*0.6, Navyblue])
   translate <shift,0,-shift>
   }
define annoying_marble2
texture {
   noise surface {
      color white
      position_fn 1
      lookup_fn 1
      octaves 3
      turbulence 3
      ambient 0.3
      diffuse 0.8
      specular 0.3
      microfacet Reitz 5
      color_map(
         [0.0, 0.8, Navyblue, Navyblue*0.6]
         [0.8, 1.0, Navyblue*0.6, Yellow])
   translate <shift,0,-shift>
   }
// create a floor
define ground
object {
   polynomial y + 0.01
   texture { checker annoying_marble1, annoying_marble2}
   scale <6, 6, 6>
   translate <0,-10,0>
// a blob explodes along the xyz axes
define framer frame/25
define dripper
object {
   blob 0.10:
      0.1, 3.0, p11*framer,
      0.1, 3.0, p12*framer,
```

```
continued from previous page
      0.1, 3.0, p13*framer,
      0.1, 3.0, p14*framer,
      0.1, 3.0, p15*framer,
      0.1, 3.0, p16*framer,
      0.1, 3.0, p21*framer,
      0.1, 3.0, p22*framer,
      0.1, 3.0, p23*framer,
      0.1, 3.0, p24*framer,
      0.1, 3.0, p25*framer,
      0.1, 3.0, p26*framer,
      0.1, 3.0, p31*framer,
      0.1, 3.0, p32*framer,
      0.1, 3.0, p33*framer,
      0.1, 3.0, p34*framer,
      0.1, 3.0, p35*framer,
      0.1, 3.0, p36*framer,
      0.5, 3.0, <0,0,0>
   root_solver Ferrari
   u steps 20
   v_steps 20
   blue_glass
dripper
define shift frame/30
ground { translate <-2*shift,0,2*shift> }
```

Some sample frames from this code are shown in Figure 2-19.

We made a few other changes. The glass in the blob was made more yellow, and its glasslike properties adjusted for higher contrast. Note that the old values were changed, but saved in // commented-out form. It's useful to either rename the file as changes are made, or save the old values, as we did







B) Frame 60 from 5DRIP1.PI

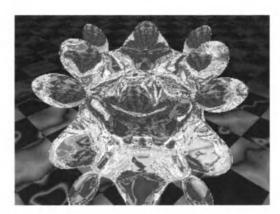


Figure 2-20 Complex blob shape during evolution

here, to keep track of where we've been, what we've tried, and provide a way to recover previous states when the changes we make go horribly wrong.

We switched from general lights to a *spot_light* that also helps to increase the contrast. We defined the checkerboard as *ground*, increased the checker size from 5 to 6, and shifted it forward a total of six units in x and z during the 90 frames. These are all minor changes, but they improved the feel of the animation.

Additional Embellishments

This last effect, as created with the following program (ANGRY.PI), will really annoy your monitor. It's derived from the Mexican title sequence from Monty Python and the Holy Grail (or Mønti Pythøn ik den Hølie Grailen, for those of you who read the original subtitles), where the colors flip from orange to yellow six times a second. Here we flip the textures on the checkerboard every single frame, positive to negative, and rather than 18 blobs along the axes, we use 60 blobs along everything *but* those axes (e.g., rather than moving along <0,0,1>, move along <1,0,1>, <1,1,0>, <0,1,1>...). The resulting mass is shown during its evolution in Figure 2-20.

```
// ANGRY.PI Angry Dripping Animation
//
// Polyray input file - Jeff Bowermaster
// define the range of the animation
start_frame 0
end_frame 89
total_frames 90
define shift frame/30
outfile angr
```

```
continued from previous page
define pi 3.14159
define rad pi/180
// set up the camera
viewpoint {
   from <10,10,-10>
   from rotate(<8,8,-8>,<0,frame,0>)+<0,2*sin(frame*rad),0>
   at <0,0,0>
   up <0,1,0>
   angle 45
   resolution 320,240
   aspect 1.33
   }
// set up background color & lights
background skyblue
define dim <0.9,0.9,0.9>
spot_light dim,<10,20,-10>,<0,0,0>, 3, 15, 30
include "\PLY\COLORS.INC"
define blue_glass
texture {
   surface {
      ambient yellow, 0.1
                                    // 0.2
      diffuse midnightblue, 0.2
                                     //0.6
      specular 0.9
                                     //0.6
      reflection white, 0.05
                                     //0.1
      microfacet Phong 2
      transmission white, 0.98, 1.5 //0.95
      }
   }
define annoying_marble1
texture {
   noise surface {
      color white
      position fn 1
      lookup_fn 1
      octaves 3
      turbulence 3
      ambient 0.3
      diffuse 0.8
      specular 0.3
      microfacet Reitz 5
      color map(
         [0.0, 0.8, Yellow, Yellow*0.6]
         [0.8, 1.0, Yellow*0.6, Navyblue])
   translate <shift,0,-shift>
```

```
}
define annoying_marble2
texture {
   noise surface {
      color white
      position_fn 1
      lookup_fn 1
      octaves 3
      turbulence 3
      ambient 0.3
      diffuse 0.8
      specular 0.3
      microfacet Reitz 5
      color map(
         [0.0, 0.8, Navyblue, Navyblue*0.6]
         [0.8, 1.0, Navyblue*0.6, Yellow])
   translate <shift,0,-shift>
// create a floor
define ground1
object {
   polynomial y + 0.01
   texture { checker annoying_marble1, annoying_marble2}
   scale <6, 6, 6>
   translate <0,-10,0>
define ground2
object {
   polynomial y + 0.01
   texture { checker annoying_marble2, annoying_marble1}
   scale <6, 6, 6>
   translate <0,-10,0>
// a blob explodes along the xyz axes
define framer frame/30
define dripper
object {
   blob 0.10:
      0.1, 3.0, < 1, 1, 0>*framer,
      0.1, 3.0, < 0, 1, 1 > * framer,
      0.1, 3.0, < 1, 0, 1>*framer,
      0.1, 3.0, <-1,-1, 0>*framer,
      0.1, 3.0, < 0, -1, -1 > * framer,
      0.1, 3.0, <-1, 0,-1>*framer,
```

CHAPTER TWO

```
continued from previous page
      0.1, 3.0, <-1, 1, 0>*framer,
      0.1, 3.0, < 0,-1, 1>*framer,
      0.1, 3.0, <-1, 0, 1>*framer,
      0.1, 3.0, < 1, -1, 0 > *framer,
      0.1, 3.0, < 0, 1, -1 > * framer,
      0.1, 3.0, < 1, 0, -1 > * framer,
      0.1, 3.0, < 1, 1, 1>*framer,
      0.1, 3.0, <-1, -1, -1 > * framer,
      0.1, 3.0, <-1, 1, 1>*framer,
      0.1, 3.0, < 1, -1, 1 > * framer,
      0.1, 3.0, < 1, 1, -1 > * framer,
      0.1, 3.0, <-1, 1,-1>*framer,
      0.1, 3.0, < 1, -1, -1 > * framer,
      0.1, 3.0, <-1,-1, 1>*framer,
      0.1, 3.0, < 2, 2, 0>*framer,
      0.1, 3.0, < 0, 2, 2 > *framer,
      0.1, 3.0, < 2, 0, 2>*framer,
      0.1, 3.0, <-2,-2, 0>*framer,
      0.1, 3.0, < 0, -2, -2 > *framer,
      0.1, 3.0, <-2, 0,-2>*framer,
      0.1, 3.0, <-2, 2, 0>*framer,
      0.1, 3.0, < 0, -2, 2 \times framer,
      0.1, 3.0, <-2, 0, 2>*framer,
      0.1, 3.0, < 2, -2, 0 > *framer,
      0.1, 3.0, < 0, 2, -2 > * framer,
      0.1, 3.0, < 2, 0, -2 * framer,
      0.1, 3.0, < 2, 2, 2>*framer,
      0.1, 3.0, <-2,-2,-2>*framer,
      0.1, 3.0, <-2, 2, 2>*framer,
      0.1, 3.0, < 2, -2, 2 \times framer,
      0.1, 3.0, < 2, 2, -2 * framer,
      0.1, 3.0, <-2, 2,-2>*framer,
      0.1, 3.0, < 2, -2, -2 \times framer
      0.1, 3.0, <-2,-2, 2>*framer,
      0.1, 3.0, < 3, 3, 0 > * framer,
      0.1, 3.0, < 0, 3, 3 > *framer,
      0.1, 3.0, < 3, 0, 3 \times framer,
      0.1, 3.0, <-3,-3, 0>*framer,
      0.1, 3.0, < 0, -3, -3 > * framer,
      0.1, 3.0, <-3, 0, -3 > * framer,
      0.1, 3.0, <-3, 3, 0>*framer,
```

0.1, 3.0, < 0, -3, 3 > * framer,

```
0.1, 3.0, <-3, 0, 3>*framer,
      0.1, 3.0, < 3, -3, 0 > *framer,
      0.1, 3.0, < 0, 3, -3 > * framer,
      0.1, 3.0, < 3, 0, -3 > * framer,
      0.1, 3.0, < 3, 3, 3>*framer,
      0.1, 3.0, <-3, -3, -3 > * framer,
      0.1, 3.0, <-3, 3, 3>*framer,
      0.1, 3.0, < 3, -3, 3 \times framer,
      0.1, 3.0, < 3, 3, -3 > * framer,
      0.1, 3.0, <-3, 3,-3>*framer,
      0.1, 3.0, < 3, -3, -3 > * framer,
      0.1, 3.0, <-3, -3, 3>*framer,
      0.5, 3.0, <0,0,0>
   root solver Ferrari
   u_steps 20
   v_steps 20
   blue_glass
dripper { rotate <0, frame,0> }
define shift frame/30
if (fmod(frame,2)==0)
   ground1 { translate <-2*shift,0,2*shift> }
else
   ground2 { translate <-2*shift,0,2*shift> }
```

As was previously mentioned, two alternative ground planes, *ground1* and *ground2*, are defined with patterns that are exact opposites; where one is black, the other is white. They'd actually be photographic negatives of each other. Depending on whether the frame counter is odd or even, one of these two grounds gets called. Alternating the grounds generates a sense (positive or negative) on the checkerboard background, giving us a flickering background.

Since the dripper blob is so detailed (60 blobs rather than 18), we decided to rotate it while it was evolving to afford a better view of the structure. It does a quarter turn during the 90 frames of this animation.

Comments

The more objects used to define a blob, the longer it takes to render, all else being equal. Refraction takes longer to render than reflection, so in developing fluid animations of your own, you might try a nontransparent texture first.



尾 2.5 How do I...

Generate a complex surface without resorting to complex triangle grids?

You'll find the code for this in: PLY\CHAPTER2\TILE

Problem

The usual way to define a complex surface like a face or a mountain range is to define a whole lot of little triangles that approximate the surface in a mosaic fashion. This takes a large data file and some expensive digitization equipment. Many vendors sell remarkably detailed 3-D models based on collections of triangles or polygons that define cows and shoes and airplanes and cities. You can probably find whatever you're looking for in some catalog, but be prepared to spend some money.

In contrast, heightfields are remarkably simple ways to get incredibly complex surfaces from fairly simple equations without having to resort to a triangle mesh. Well, you *personally* don't have to resort to them—the ray tracer generates triangles that it uses internally, so they become its problem, not yours.

Technique

In Section 2.1, we used a heightfield for the terrain. We'll use another one here, one with shifting walls and alternating patches of smooth (slowly varying) and noisy (rapidly varying) heights. It's based on raising the height of a point on a surface by an exponent generated by dividing the cosine of its x location by the sine of its z location. For those of you having problems visualizing this, it's shown in Figure 2-21.



Figure 2-21 A surface defined by $x^{(\cos(x)/\sin(z))} + z^{(\sin(x)/\cos(z))}$ overlaid by grid

At certain points, the surface easily extends well beyond the next galaxy, so we make use of *fmods* (floating point modulus... remainder after division by some arbitrary number) to yank it back onto our monitor. We also scale the height of the surface so that it rises and falls like the surface of the sea. To heighten the sense of rising and falling, we'll add a stable grid of spheres that will act like a pier in this sea. These spheres are placed at 50% of the maximum height of the sea, which allows us to see how much the surface as a whole is moving up and down. We start by staring at a small region on the surface, then graduate to a flyover.

Steps

This first animation (TILE1.PI) focuses in on a single region on the surface. The camera is controlled by a tricornered function (a deltoid) centered on a square where a reasonably interesting surface evolution was found during some test renders. The grid size of the heightfield, which determines how many triangles are laid down, is made to vary from frame to frame, resulting in changes in spatial resolution and shifting features. The height is accented by a 3-D yellow and blue checker function. It's been scaled and positioned to hide the fact that it's a checker. Checkers are overused textures in ray traced images and should be avoided whenever possible. A grid function lays down some lines of spheres marking borders between smooth and noisy areas, and giving a better reference to the large scale height shifts in the smoother areas on the heightfield.

```
// TILE1.PI
// Heightfield Tiled Function
// Polyray input file: Jeff Bowermaster
start_frame 0
end_frame 359
total_frames 360
OUTFILE TILE1
define angle_nor frame/total_frames
define pi 3.14159
define pi2 2*pi
define rad pi/180
define a 2
define b 0.5
define phz1 0
define phz2 0.25
define sides 3-1
// two Deltoids
```

```
continued from previous page
// the view from x & z points
define vx pi * COS((angle_nor + phz1) * pi2)
define vz pi2 * SIN((angle_nor + phz1) * pi2)
// the look at x and z points
define lx sides * b * COS((angle_nor + phz2) * pi2) + b * COS(sides *
(angle_nor + phz2) * pi2)
define lz sides * b * SIN((angle_nor + phz2) * pi2) - b * SIN(sides *
(angle_nor + phz2) * pi2)
define height 3 + 2*cos(angle_nor*pi2)
viewpoint {
   from <1.5*pi+vx,height,-1*pi+vz>
   at <1.5*pi+lx,0,-1*pi+lz>
   up <0,1,0>
   angle 30
   resolution 320,200
   aspect 1.43
include "\PLY\COLORS.INC"
// position the lights on a circle radius 40 that rotates once
define xl1 40*cos(angle_nor*pi2)
define zl1 40*sin(angle_nor*pi2)
define xl2 40*cos((angle_nor+0.333)*pi2)
define zl2 40*sin((angle_nor+0.333)*pi2)
define xl3 40*cos((angle_nor+0.666)*pi2)
define zl3 40*sin((angle_nor+0.666)*pi2)
// tricolor orbiting lights
light <1,0.4,0.2>,<xl1,20,zl1> // mostly red
light <0.4,1,0.2>,<xl2,20,zl2> // mostly green
light <0.2,0.4,1>,<xl3,20,zl3>
                                 // mostly blue
// the function "scaler" modulates the surface between smooth and noisy
// it's shaped like the universal symbol for water:
//
11
//
//
11
11
//
//
11
```

```
// with values between 0.5 and 6.0
define scaler 6.5-(18.25 - 18 * SIN(angle nor * pi2 )) ^ 0.5
define a sin(x)/cos(z)
define b cos(z)/sin(x)
define HFn fmod(|x|^a + |z|^b, scaler)/scaler
define detail 100*scaler
// define a 3-D checkerboard surface
object {
   smooth_height_fn detail, detail, 0, 3*pi, -3*pi, pi, HFn
   texture {
      checker shiny_yellow, shiny_blue
      translate <0, -0.1, 0.5>
      scale <pi, 0.1, pi>
   }
// lay down a grid of spheres halfway up to show off height motion
define ball1 object { sphere <0.0, 0.0, 0.0>, 1 shiny_blue}
define ball2 object { sphere <0.0, 0.0, 0.0>, 1 shiny_coral}
define ball3 object { sphere <0.0, 0.0, 0.0>, 1 mirror}
define snf pi/25
object {
   gridded "gridmask.tga",
      ball1
      ball2
      ball3
   translate <-50, 0, -50>
   scale <snf, snf, snf>
   translate <pi2, 0.5, -pi2>
```

How It Works

The equations that define where the camera is and what it looks at are shown graphically in Figure 2-22. The camera moves in an oval path around the interesting area, and the focus of the camera paints a deltoid pattern on the surface. The height of the camera varies from one to five units above the plane.

We define three colored lights that orbit 120° apart 20 units above the plane in a circle of radius=40. This produces shifting pools of varying colors that make the shadows more interesting.

The *scaler* function, described with ASCII graphics in the comments, causes the surface height to rise and fall like the surface of the sea. The heightfield function

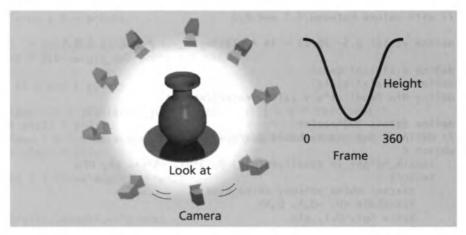


Figure 2-22 The camera motion, what it looks at, and its height during the animation

```
define a sin(x)/cos(z)
define b cos(z)/sin(x)
define HFn fmod(|x|^a + |z|^b,scaler)/scaler
```

is odd, but it has some periodicity with respect to pi. This periodicity is exploited to hide the checkerboard nature of the texture map, which we apply to color the surface different colors at different heights, and to give the map a pseudo-topographic look.

The final element in the image is a gridded object, our reference plane, which is generated by reading a targa file and putting a colored sphere into our image wherever a pixel of the same color exists in the targa image (Figure 2-23). We could have just generated an include file full of spheres in

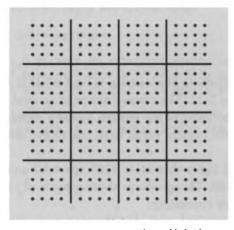


Figure 2-23 GRIDMASK.TGA, the gridded reference map for our graph paper in the image

the appropriate places, but this kind of graph paper reference grid is exactly what gridded objects are good at, and learning something new about Polyray is always worthwhile.

TILE.PI is a bit tough to watch for any length of time, because it rocks and surges quite a bit. Though really just a simple oval flyby, TILE.PI creates a more pleasing animation by eliminating spatial resolution shifts and moving close to the surface.

```
// TILE2.PI
// Heightfield Tiled Function
// Polyray input file: Jeff Bowermaster
start_frame 0
end_frame 359
total_frames 360
outfile "tile2"
define angle_nor frame/total_frames
define pi 3.14159
define pi2 2*pi
define rad pi/180
define a 2
define b 0.5
define phz1 0
define phz2 0.25
define sides 3-1
define vx pi * COS((angle_nor + phz1) * pi2)
define vz pi2 * SIN((angle_nor + phz1) * pi2)
define lx sides * b * COS((angle nor + phz2) * pi2) + b * COS(sides * (angle nor + phz2) * pi2)
define lz sides * b * SIN((angle_nor + phz2) * pi2) - b * SIN(sides * (angle_nor + phz2) * pi2)
define height 3 + 2*cos(angle_nor*pi2)
viewpoint {
   from <1.5*pi+vx,height,-1*pi+vz>
   at <1.5*pi+lx_0,-1*pi+lz>
   up <0,1,0>
   angle 30
   resolution 320,200
   aspect 1.43
include "\PLY\COLORS.INC"
// position the lights on a circle radius 40 that rotates once
                                                                     continued on next page
```

CHAPTER TWO

```
continued from previous page
define xl1 40*cos(angle_nor*pi2)
define zl1 40*sin(angle_nor*pi2)
define xl2 40*cos((angle nor+0.333)*pi2)
define zl2 40*sin((angle_nor+0.333)*pi2)
define xl3 40*cos((angle_nor+0.666)*pi2)
define zl3 40*sin((angle_nor+0.666)*pi2)
light <1,0.4,0.2>,<xl1,20,zl1>
                                 // tricolor orbiting lights
light <0.4,1,0.2>,<xl2,20,zl2>
light <0.2,0.4,1>,<xl3,20,zl3>
define scaler 6.5-(18.25 - 18 * SIN(angle_nor * pi2 )) ^ 0.5
define a sin(x)/cos(z)
define b cos(z)/sin(x)
define HFn fmod(|x|^a + |z|^b, scaler)/scaler
define detail 250
// define a 3-D checkerboard surface
object {
   smooth_height_fn detail, detail, 0, 3*pi, -3*pi, pi, HFn
   texture {
      checker shiny_yellow, shiny_blue
      translate <0, -0.1, 0.5>
      scale <pi, 0.1, pi>
      }
   }
// lay down a grid of spheres to pick reasonable fly points
// make an array of fiducial spheres
define ball1 object { sphere <0.0, 0.0, 0.0>, 1 shiny_blue}
define ball2 object { sphere <0.0, 0.0, 0.0>, 1 shiny_coral}
define ball3 object { sphere <0.0, 0.0, 0.0>, 1 mirror}
define snf pi/25
object {
   gridded "gridmask.tga",
      ball1
      ball2
      ball3
   translate <-50, 0, -50>
   scale <snf, snf, snf>
   translate <pi2, 0.5, -pi2>
```



Generate tumbling motions?

You'll find the code for this in: PLY\CHAPTER2\DICE

Problem

Examining an object from all sides isn't as easy as it sounds. Constant rotations about an axis, besides being rather plain, don't bring every side of an object into view. And constant rotation about several axes just adds together to create a simple rotation about a composite axis. So the trick here (surprise, surprise) is *not* to use constant rotations, but to vary the rotational speed and phase about several axes, and some combination will eventually make every side of an object come around to the front. One type of motion that fits the bill is the stitch pattern on a baseball.

Technique

What would you do if someone handed you a diamond, I mean a really HUGE diamond? You'd run away with it, of course (silly question). No, no no, let's say there are guards with guns and poor social skills staring at you with beady little eyes below protruding foreheads. OK, you'll just have to be satisfied with looking at it. What would you do then; would you hold it at arm's length and just stare at from one angle? Of course not! You'd tumble it end over end, upside down, right side up, inside out (tough, by the way), backwards, forwards, you'd try to look at that thing from every conceivable angle. Fine. You get the idea. That's just what we're going to try to do right now.

In order to demonstrate that this motion indeed fits the bill, bringing all sides into view without bias, we approach it analytically. Figure 2-24 shows in patented crypto-form (no labels on any axis) the x, y, and z angular displacements, the sum of the distances of each vertex from its original orientation (used to find loop points), histograms (sums as bar charts) of the number of frames each corner and each face spends in the foreground (there's a difference), and the cube as it tumbles. The goal here is to find a set of motions that generates flat histograms for the faces or corners, meaning that all faces or corners spend about the same amount of time in the foreground. The tumbling equations that produced this result are

```
xrotate = factor * (2 * SIN(ang) + SIN(3 * ang) / 3)
yrotate = factor * (2 * COS(ang) - COS(3 * ang) / 3)
zrotate = factor * (COS(2 * ang))
```

These cover a range for any of 0 to 360.

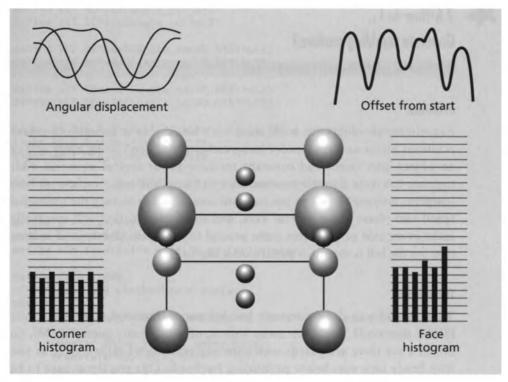


Figure 2-24 Statistical treatment of a generalized object tumbling routine

Steps

Figure 2-24 was generated with the following code (TUMBLE.BAS). It not only writes the preceding data to the screen, it also writes a file called MARKER.INC, which contains the path represented by a series of spheres that a stick imbedded in the cube would trace out around the cube as it tumbled.

```
' TUMBLE.BAS
```

DECLARE SUB rotate (x, y, z)
COMMON SHARED rad, a, xrotate, yrotate, zrotate

TYPE Vector

- x AS SINGLE
- y AS SINGLE
- z AS SINGLE

END TYPE

DIM cube(8) AS Vector, c(8) AS Vector, last(8) AS Vector, hist(8)

```
DIM f(8) AS Vector, fo(8) AS Vector, r(8) AS Vector, marker AS Vector
' set the screen up with pretty rainbow colors
SCREEN 12
WINDOW (-4, -2.2)-(4, 3.8)
pi = 3.1415926535#
rad = pi / 180
' A static reference cube
FOR v = 1 TO 8
       READ c(v).x, c(v).y, c(v).z
NEXT v
DATA 1, 1, 1
DATA 1, 1, -1
DATA 1, -1, 1
DATA 1, -1, -1
DATA -1, 1, 1
DATA -1, 1, -1
DATA -1, -1, 1
DATA -1, -1, -1
OPEN "\marker.inc" FOR OUTPUT AS #1
'DO WHILE INKEY$ = ""
FOR a = -1 TO 1.5 STEP .1
       LINE (-3.15, a)-(-2.48, a)
       LINE (2.48, a)-(3.15, a)
NEXT a
LOCATE 26, 10: PRINT "Corner"
LOCATE 27, 8: PRINT "Histogram"
LOCATE 26, 67: PRINT "Face"
LOCATE 27, 65: PRINT "Histogram"
FOR a = 1 TO 8
       cube(a) = c(a)
NEXT a
minang = 0
mindiff = 20
FOR angle = 0 \text{ TO } 360
       an = (angle - 360) / 720
       FOR a = 1 TO 8
               cube(a) = c(a)
       NEXT a
       ang = angle * rad
```

```
continued from previous page
       factor = 90
       xrotate = factor * (2 * SIN(ang) + SIN(3 * ang) / 3)
       yrotate = factor * (2 * COS(ang) - COS(3 * ang) / 3)
       zrotate = factor * (COS(2 * ang))
       marker.x = 3
       marker.y = 3
       marker.z = 3
       CALL rotate(marker.x, marker.y, marker.z)
       FOR a = 1 TO 8
               ' semi-equal-coverage tumbling
               xo = -1.3
               yo = 2.5
              PSET (4 * an + xo, xrotate / 360 + yo), 4
               PSET (4 * an + xo, yrotate / 360 + yo), 2
               PSET (4 * an + xo, zrotate / 360 + yo), 1
'rotate
               CALL rotate(cube(a).x, cube(a).y, cube(a).z)
               IF (angle = 0) THEN
                 r(a).x = cube(a).x
                 r(a).y = cube(a).y
                 r(a).z = cube(a).z
         END IF
        NEXT a
'undraw the old ones
        FOR a = 1 TO 8
                CIRCLE (last(a).x, last(a).y), (last(a).z + 5) / 15, 0
        NEXT a
        FOR a = 1 TO 6
                CIRCLE (fo(a).x, fo(a).y), .1, 0
        NEXT a
        LINE (last(1).x, last(1).y)-(last(2).x, last(2).y), 0
        LINE (last(2).x, last(2).y)-(last(4).x, last(4).y), 0
        LINE (last(3).x, last(3).y)-(last(1).x, last(1).y), 0
        LINE (last(4).x, last(4).y)-(last(3).x, last(3).y), 0
        LINE (last(5).x, last(5).y)-(last(6).x, last(6).y), 0
        LINE (last(6).x, last(6).y)-(last(8).x, last(8).y), 0
        LINE (last(8).x, last(8).y)-(last(7).x, last(7).y), 0
```

```
LINE (last(7).x, last(7).y)-(last(5).x, last(5).y), 0
        LINE (last(1).x, last(1).y)-(last(5).x, last(5).y), 0
        LINE (last(2).x, last(2).y)-(last(6).x, last(6).y), 0
        LINE (last(3).x, last(3).y)-(last(7).x, last(7).y), 0
        LINE (last(4).x, last(4).y)-(last(8).x, last(8).y), 0
'draw the new ones
        maxz = 0
        FOR a = 1 TO 8
                'CIRCLE (cube(a).x, cube(a).y), (cube(a).z + 5) / 15, \leftarrow
INT((cube(a).z + 2) * 4)
               CIRCLE (cube(a).x, cube(a).y), (cube(a).z + 5) / 15, a + 1
        IF a = 1 THEN PRINT #1, USING "##.##### ##.##### ##.#####"; ←
cube(a).x, cube(a).y, cube(a).z
                last(a) = cube(a)
                IF (cube(a).z) > maxz THEN
                       maxz = cube(a).z
                       n = a
                END IF
        NEXT a
        hist1(n) = hist1(n) + 1
' average z values for each face
        f(1).x = (cube(1).x + cube(2).x + cube(3).x + cube(4).x) / 4
        f(2).x = (cube(5).x + cube(6).x + cube(7).x + cube(8).x) / 4
        f(3).x = (cube(1).x + cube(2).x + cube(5).x + cube(6).x) / 4
        f(4).x = (cube(1).x + cube(3).x + cube(5).x + cube(7).x) / 4
        f(5).x = (cube(3).x + cube(4).x + cube(7).x + cube(8).x) / 4
        f(6).x = (cube(2).x + cube(4).x + cube(6).x + cube(8).x) / 4
        f(1).y = (cube(1).y + cube(2).y + cube(3).y + cube(4).y) / 4
        f(2).y = (cube(5).y + cube(6).y + cube(7).y + cube(8).y) / 4
        f(3).y = (cube(1).y + cube(2).y + cube(5).y + cube(6).y) / 4
        f(4).y = (cube(1).y + cube(3).y + cube(5).y + cube(7).y) / 4
        f(5).y = (cube(3).y + cube(4).y + cube(7).y + cube(8).y) / 4
        f(6).y = (cube(2).y + cube(4).y + cube(6).y + cube(8).y) / 4
        f(1).z = (cube(1).z + cube(2).z + cube(3).z + cube(4).z) / 4
        f(2).z = (cube(5).z + cube(6).z + cube(7).z + cube(8).z) / 4
        f(3).z = (cube(1).z + cube(2).z + cube(5).z + cube(6).z) / 4
        f(4).z = (cube(1).z + cube(3).z + cube(5).z + cube(7).z) / 4
        f(5).z = (cube(3).z + cube(4).z + cube(7).z + cube(8).z) / 4
        f(6).z = (cube(2).z + cube(4).z + cube(6).z + cube(8).z) / 4
                                                                  continued on next page
```

```
continued from previous page
        maxz = -1
        FOR a = 1 TO 6
                CIRCLE (f(a).x, f(a).y), .1, a + 1
                fo(a).x = f(a).x
                fo(a).y = f(a).y
                fo(a).z = f(a).z
                IF (f(a).z) > maxz THEN
                       maxz = f(a).z
                       n = a
                END IF
        NEXT a
        hist2(n) = hist2(n) + 1
        LINE (cube(1).x, cube(1).y)-(cube(2).x, cube(2).y), 15
        LINE (cube(2).x, cube(2).y)-(cube(4).x, cube(4).y), 15
        LINE (cube(3).x, cube(3).y)-(cube(1).x, cube(1).y), 15
        LINE (cube(4).x, cube(4).y)-(cube(3).x, cube(3).y), 15
        LINE (cube(5).x, cube(5).y)-(cube(6).x, cube(6).y), 15
        LINE (cube(6).x, cube(6).y)-(cube(8).x, cube(8).y), 15
        LINE (cube(8).x, cube(8).y)-(cube(7).x, cube(7).y), 15
        LINE (cube(7).x, cube(7).y)-(cube(5).x, cube(5).y), 15
        LINE (cube(1).x, cube(1).y)-(cube(5).x, cube(5).y), 15
        LINE (cube(2).x, cube(2).y)-(cube(6).x, cube(6).y), 15
        LINE (cube(3).x, cube(3).y)-(cube(7).x, cube(7).y), 15
        LINE (cube(4).x, cube(4).y)-(cube(8).x, cube(8).y), 15
        LOCATE 1, 1
        offx1 = 40
        offy1 = 100
        offx2 = -30
        offy2 = 100
        scale = .04
        diff = 0
                        ' sum of the distances from original orientation
        FOR a = 1 TO 8
                x1 = (a - offx1) * .08
                y1 = (hist1(a) - offy1) * .01
                x2 = (a - offx2) * .08
                y2 = (hist2(a) - offy2) * .01
                diff = diff + ((r(a).x - cube(a).x) ^ 2 + (r(a).y - cube(a).y) \Leftarrow
^ 2 + (r(a).z - cube(a).z) ^ 2) ^ .5
                LINE (x1, y1)-(x1 + scale, y1 + scale), a + 1, BF
                IF a < 7 THEN LINE (x2, y2)-(x2 + scale, y2 + scale), a + 1, BF
                COLOR a + 1
```

```
PRINT USING "### "; hist1(a);
        NEXT a
        LOCATE 2, 1
        FOR a = 1 TO 6
                COLOR a + 1
                PRINT USING "### "; hist2(a);
        NEXT a
        PRINT
        COLOR 15
        IF angle > 100 THEN
                IF mindiff > diff THEN
                       mindiff = diff
                       minang = angle
                END IF
        END IF
        PRINT USING "#### "; angle, xrotate, yrotate, zrotate
        PRINT #1, USING "if (frame > ###) { object { sphere ←
<###.##,###.##,###.##>, 0.25 shiny_yellow }}"; angle, marker.x, marker.y, \(\infty\)
marker.z
        CIRCLE (4 * an + 2, diff / 20 + 2), .01, 12
  NEXT angle
CLOSE #1
SUB rotate (x, y, z)
               x = 0x
               y0 = y
               z0 = z
               x1 = x0
               y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
               z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)
               x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
               y2 = y1
               z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)
               x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)
               y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)
               z3 = z2
               x = x3
               y = y3
               z = z3
```

END SUB

Doug Reedy did a whole series of tumbling dice animations. DICE.PI uses one of his dice, glitzed up a bit for impact, to show the kind of tumbling

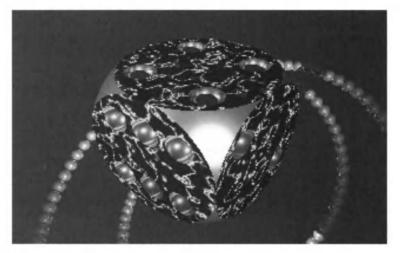


Figure 2-25 Die with trail showing where it's been

motion we have been discussing. This program includes the MARKER.INC file, that generates the trail of bread crumbs, as it were, to show the path of the tumbling cube (Figure 2-25). It turns out that it's not a very symmetrical path, but histograms don't lie, or at least they didn't used to.

```
// DICE.PI
start_frame 0
end_frame 359
total_frames 360
outfile "dice"
viewpoint {
   from <8, 8, -8>
   at <0,0,0>
   up <0,1,0>
   angle 40
   resolution 64,40
   aspect 1.433
background SkyBlue
include "\PLY\COLORS.INC"
spot_light white, <-5,10,-5>,<0,0,0>,3,5,20
spot_light white, < 5,10,-5>,<0,0,0>,3,5,20
define die1
   object {
      object {box <-2,-2,-2>,<2,2,2> }
```

```
//1
    - object {sphere < 0, 0,-2>, 0.4 matte_black }
    - object {sphere <-2,-1,-1>, 0.4 matte_black }
    - object {sphere <-2, 1, 1>, 0.4 matte_black }
    - object {sphere <-1,-2,-1>, 0.4 matte_black }
    - object {sphere < 0,-2, 0>, 0.4 matte_black }
    - object {sphere < 1,-2, 1>, 0.4 matte_black }
    - object {sphere <-1, 2,-1>, 0.4 matte_black }
    - object {sphere < 1, 2,-1>, 0.4 matte_black }
    - object {sphere <-1, 2, 1>, 0.4 matte_black }
    - object {sphere < 1, 2, 1>, 0.4 matte_black }
   //5
    - object {sphere < 2, 0, 0>, 0.4 matte_black }
   - object {sphere < 2,-1,-1>, 0.4 matte_black }
   - object {sphere < 2, 1, 1>, 0.4 matte_black }
    - object {sphere < 2,-1, 1>, 0.4 matte_black }
    - object {sphere < 2, 1,-1>, 0.4 matte_black }
   //6
    - object {sphere <-1, 1, 2>, 0.4 matte_black }
    - object {sphere < 0, 1, 2>, 0.4 matte_black }
    - object {sphere < 1, 1, 2>, 0.4 matte_black }
    - object {sphere <-1,-1, 2>, 0.4 matte_black }
    - object {sphere < 0,-1, 2>, 0.4 matte_black }
    - object {sphere < 1,-1, 2>, 0.4 matte_black }
define dice
object {
  die1 { rotate <45,45,0> }
* object { sphere < 0, 0, 0>, 2.85 }
define pi 3.14159
define rad pi/180
define ang frame * rad
define xrotate 90 * (2 * SIN(ang) + SIN(3 * ang) / 3)
define yrotate 90 * (2 * COS(ang) - COS(3 * ang) / 3)
define zrotate 90 * (0.9 * COS(2 * ang))
dice { rotate <xrotate,yrotate,zrotate> }
include "marker.inc"
```

How It Works

The die is defined as a box with a series of patterned black spheres cut out of it using the constructive solid geometry (CSG) difference (-) operator. Constructive solid geometry is a tool that allows you to contruct complex

objects as a collection of simpler geometric forms, like boxes and spheres, and use math and logic to add, subtract and clip the parts into the desired shape. Look for more details in the Polyray documentation. The edges of the dice are smoothed by showing the intersection (*) of the box and a sphere. A die is defined, then called and rotated.

MARKER.INC, the following listing, contains a long series of IF statements that, based on the frame count, show where the cube has been up to the current frame. There are 360 IF statements, and, depending on the frame count, a sphere either gets shown or it doesn't. As is the case with all long scripts in this book, only a portion of the code is shown, and it is always generated programmatically by QuickBasic, never manually.

```
// Marker.inc
             0) { object { sphere < -4.00, -1.46, -5.46>, 0.25 reflective_white }}
if (frame >
             1) { object { sphere < -3.66, -1.31, -5.74>, 0.25 reflective_white }}
if (frame >
             2) { object { sphere < -3.29, -1.17, -5.98>, 0.25 reflective_white }}
if (frame >
             3) { object { sphere < -2.91, -1.04, -6.20>, 0.25 reflective_white }}
if (frame >
if (frame >
             4) { object { sphere < -2.50, -0.94, -6.39>, 0.25 reflective_white }}
             5) { object { sphere < -2.08, -0.86, -6.55>, 0.25 reflective_white }}
if (frame >
             6) { object { sphere < -1.65, -0.80, -6.68>, 0.25
if (frame >
. . .
```

Rotations Versus Viewpoints

It's tough finding reasonable rotational functions for objects. It's much easier to generate a viewpoint that crawls all over an object. Two equations come in quite handy for this sort of thing. A three-phased ramp function, as shown

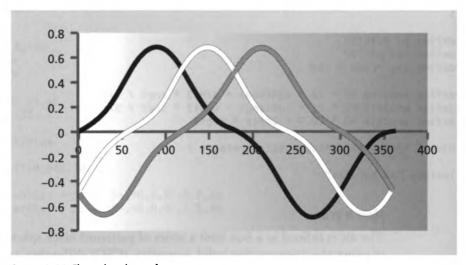


Figure 2-26 Three-phased ramp function

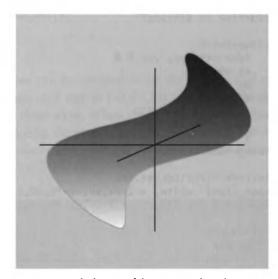


Figure 2-27 The location of the camera under a threephased ramp function

in Figure 2-26, can be generated using SIN(angle) - SIN(3*angle) / 5. It sequentially focuses on all six sides of a die. The second SIN term makes the wave form triangular, giving it that extra snap. The 3-D locations of the camera are shown in Figure 2-27. In this next iteration of the tumbling dice (DICE3.PI), the three-phased ramp function is incorporated.

```
//DICE3.PI
start_frame O
end_frame 59
total_frames 60
outfile "dice"
define rate 360/total frames
define pi 3.14159
define rad pi/180
define ang1 (frame*rate + 0) * rad
define ang2 (frame*rate + 60) * rad
define ang3 (frame*rate + 120) * rad
define vx SIN(ang1) - SIN(3 * ang1) / 5
define vy SIN(ang2) - SIN(3 * ang2) / 5
define vz SIN(ang3) - SIN(3 * ang3) / 5
//define vx SIN(ang1)
//define vy SIN(ang2)
```

```
continued from previous page
//define vz SIN(ang3)
viewpoint {
   from \langle vx, vy, vz \rangle * 8
   at <0,0,0>
   up <0,1,0>
   angle 40
   resolution 160,100
   aspect 1.433
background SkyBlue
include "\PLY\COLORS.INC"
spot_light white, <-2+vx,vy,vz>*8,<0,0,0>,3,5,20
spot_light white, < 2+vx,vy,vz>*8,<0,0,0>,3,5,20
define die1
   object {
      object {box <-2,-2,-2>,<2,2,2> matte_white}
    - object {sphere < 0, 0,-2>, 0.4 matte_black }
   //2
    - object {sphere <-2,-1,-1>, 0.4 matte_black }
    - object {sphere <-2, 1, 1>, 0.4 matte_black }
    - object {sphere <-1,-2,-1>, 0.4 matte_black }
    - object {sphere < 0,-2, 0>, 0.4 matte_black }
    - object {sphere < 1,-2, 1>, 0.4 matte_black }
   1/4
    - object {sphere <-1, 2,-1>, 0.4 matte_black }
    - object {sphere < 1, 2,-1>, 0.4 matte_black }
    - object {sphere <-1, 2, 1>, 0.4 matte_black }
    - object {sphere < 1, 2, 1>, 0.4 matte_black }
   1/5
    - object {sphere < 2, 0, 0>, 0.4 matte_black }
    - object {sphere < 2,-1,-1>, 0.4 matte_black }
    - object {sphere < 2, 1, 1>, 0.4 matte_black }
    - object {sphere < 2,-1, 1>, 0.4 matte_black }
    - object {sphere < 2, 1,-1>, 0.4 matte_black }
   116
    - object {sphere <-1, 1, 2>, 0.4 matte_black }
    - object {sphere < 0, 1, 2>, 0.4 matte_black }
    - object {sphere < 1, 1, 2>, 0.4 matte_black }
    - object {sphere <-1,-1, 2>, 0.4 matte_black }
    - object {sphere < 0,-1, 2>, 0.4 matte_black }
    - object {sphere < 1,-1, 2>, 0.4 matte_black }
define dice
object {
   die1
```

```
* object { sphere < 0, 0, 0>, 2.85 matte_white} } dice
```

For less snap, the second term can be omitted from the vx, vy, and vz equations, and the results (commented out in DICE3.PI) produce a simple orbital motion, tilted 45° to all three axes. When there is little else in your scene besides the one object, zooming the camera around the object is just as effective as holding the camera steady and tumbling the object. Some frames from DICE3.PI are shown in Figure 2-28.

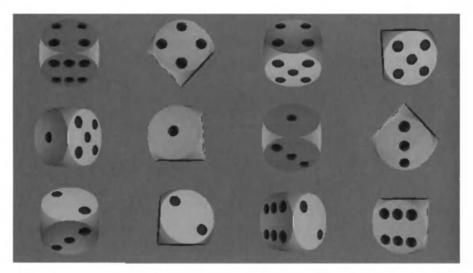


Figure 2-28 Frames from DICE3.PI

BARMER

3

POINTS OF VIEW

s is true with most things, perception depends on your point of view. In this chapter, we'll show animations where we either shift the point of view to reveal some hidden facet of an image, or create an image that is not what it seems. Shifting around off-camera elements—for example, random wandering shapes intersecting colored spotlights at various levels—can produce unusual textures that would be hard to generate on the objects themselves. Camera motions can transform flat, 2-D images into dramatic 3-D environments that pull the viewer right into the screen. The underlying theme here is that clever tricks can make complex scenes from simple data files, and that moving around really adds reality to the spatial feel of animations.



3.1 How do I...

Create a dazzling background with moving bands of color?

You'll find the code for this in: PLY\CHAPTER3\BARCODE

Problem

Let's say you need a vibrant background with randomly shifting color bands. Polyray has a "noise" surface texture with several linear color mapping functions that would work. We'll cover those in the next section. Generating randomly changing color maps to feed these textures could be a real chore. As an alternative, we'll use colored lights filtered through shifting off-camera elements to generate the same effect with less work.

Technique

In the main library of North Carolina State University, there was (and might still be) an interesting sculpture where a series of slats mounted a short distance from a wall were illuminated by an array of blinking colored lights. The lights were sequenced in a semi-random fashion, and formed a giant, dynamic Technicolor bar code on the wall. The colored patches were a combination of both the lights themselves and the complementary colors formed as a result of the way eyes deal with the shadows of colored lights. It's really easy to model this with a ray tracer.

Writing the Program

This animation requires separate controls for a great number of objects. Rather than come up with separate functions to control each element, we'll use a sine function and space our variables out along this single function.

In Figure 3-1, there are 11 spotlights pointing towards a wall with an equal number of cylinders (seen end-on) intersecting their light from behind a camera. Each spot has its own color. Some oscillating spheres traveling across the scene are added and made shiny to show off what's going on behind the scenes. The camera points at the wall and can't see what the cylinders are doing. From this perspective, we'll move the spots left and right, the cylinders left and right, and the oscillating spheres top to bottom.

Everything shifts: the position of the lights, their intensity, the size of the oscillating spheres, the positions of the blocking cylinders. All is done with a *sine* function indexed to the normalized frame count (*frameltotal_frames*). This makes the animation smoothly loop at the end, as shown in BARCODE.PI—the next listing.

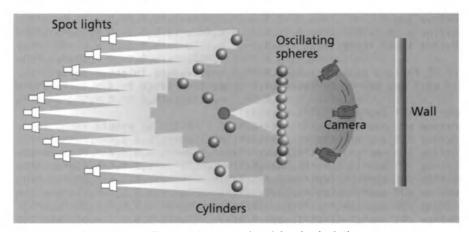


Figure 3-1 Lights intersecting off-camera elements produce shifting bands of color

```
//BARCODE.PI - Colored Lights and Tubes
start_frame 0
end_frame 59
total_frames 60
outfile "barc"
include "\PLY\COLORS.INC"
viewpoint {
   from <0,0,1>
   at <0,0,5>
   up <0,1,0>
   angle 45
   resolution 320,200
   aspect 1.43
background MidnightBlue
// 11 different colors
define c01 <1.000,0.500,0.000>
                                      //coral
define c02 <0.500,0.500,0.000>
                                      //yellow
define c03 <1.000,0.000,0.500>
                                      //red purple
                                      //blue violet
define c04 <0.500,0.000,1.000>
define c05 <0.000,1.000,0.500>
                                      //green blue
define c06 <0.000,0.500,1.000>
                                      //blue green
define c07 <1.000,0.000,0.000>
                                      //red
define c08 <1.000,0.250,0.000>
                                      //orange
define c09 <0.000,0.000,1.000>
                                      //blue
define c10 <1.000,0.000,1.000>
                                      //magenta
define c11 <0.000,0.250,1.000>
                                      //aqua
```

CHAPTER THREE

```
continued from previous page
define amp 2
define pi 3.14159
define index frame/total_frames
// 11 factors equally spaced 1/11th units apart in phase space
// with amp set at 2, these will range from -2 to 2
define a01 amp*sin(2*pi*(index+1/11))
define a02 amp*sin(2*pi*(index+2/11))
define a03 amp*sin(2*pi*(index+3/11))
define a04 amp*sin(2*pi*(index+4/11))
define a05 amp*sin(2*pi*(index+5/11))
define a06 amp*sin(2*pi*(index+6/11))
define a07 amp*sin(2*pi*(index+7/11))
define a08 amp*sin(2*pi*(index+8/11))
define a09 amp*sin(2*pi*(index+9/11))
define a10 amp*sin(2*pi*(index+10/11))
define all amp*sin(2*pi*(index+11/11))
// these will go from 0 to 2
define b01 (a01+amp)/amp
define b02 (a02+amp)/amp
define b03 (a03+amp)/amp
define b04 (a04+amp)/amp
define b05 (a05+amp)/amp
define b06 (a06+amp)/amp
define b07 (a07+amp)/amp
define b08 (a08+amp)/amp
define b09 (a09+amp)/amp
define b10 (a10+amp)/amp
define b11 (a11+amp)/amp
// use the factors above, but make them go from 0 to 2
// dim lets us simultaneously reduces all the lights
// multiple lights usually overexpose RT scenes
define dim 0.5
// 11 colored spot_lights, pointing along the z-axis. Their
// distance from the wall changes from -8 to -12, and varies
// in brightness from double their nominal brightness to off
define dist -10 // how far back the spots are
define strt 5 // inner angle for spotlight
define fini 10 // outer angle ""
// these are the lower lights (y = -2.8)
spot_light c01*dim*b08,<-5,-2.8,dist+a01>,<-5,-2.8,0>,3,strt,fini
spot_light c02*dim*b05,<-4,-2.8,dist+a02>,<-4,-2.8,0>,3,strt,fini
spot_light c03*dim*b01,<-3,-2.8,dist+a03>,<-3,-2.8,0>,3,strt,fini
spot_light c04*dim*b03,<-2,-2.8,dist+a04>,<-2,-2.8,0>,3,strt,fini
```

```
spot_light c05*dim*b11,<-1,-2.8,dist+a05>,<-1,-2.8,0>,3,strt,fini
spot_light c06*dim*b02,< 0,-2.8,dist+a06>,< 0,-2.8,0>,3,strt,fini
spot light c07*dim*b10,< 1,-2.8,dist+a07>,< 1,-2.8,0>,3,strt,fini
spot_light c08*dim*b04,< 2,-2.8,dist+a08>,< 2,-2.8,0>,3,strt,fini
spot_light c09*dim*b09,< 3,-2.8,dist+a09>,< 3,-2.8,0>,3,strt,fini
spot light c10*dim*b06,< 4,-2.8,dist+a10>,< 4,-2.8,0>,3,strt,fini
spot_light c11*dim*b07,< 5,-2.8,dist+a11>,< 5,-2.8,0>,3,strt,fini
// these are the upper lights (y = 2.8)
spot_light c01*dim*b08,<-5,2.8,dist+a01>,<-5,2.8,0>,3,strt,fini
spot_light c02*dim*b05,<-4,2.8,dist+a02>,<-4,2.8,0>,3,strt,fini
spot light c03*dim*b01,<-3,2.8,dist+a03>,<-3,2.8,0>,3,strt,fini
spot_light c04*dim*b03,<-2,2.8,dist+a04>,<-2,2.8,0>,3,strt,fini
spot_light c05*dim*b11,<-1,2.8,dist+a05>,<-1,2.8,0>,3,strt,fini
spot light c06*dim*b02,< 0,2.8,dist+a06>,< 0,2.8,0>,3,strt,fini
spot_light c07*dim*b10,< 1,2.8,dist+a07>,< 1,2.8,0>,3,strt,fini
spot_light c08*dim*b04,< 2,2.8,dist+a08>,< 2,2.8,0>,3,strt,fini
spot_light c09*dim*b09,< 3,2.8,dist+a09>,< 3,2.8,0>,3,strt,fini
spot_light c10*dim*b06,< 4,2.8,dist+a10>,< 4,2.8,0>,3,strt,fini
spot_light c11*dim*b07,< 5,2.8,dist+a11>,< 5,2.8,0>,3,strt,fini
// The intersecting cylinders we use to create the shadows
object { cylinder <-5,-5,0>,<-5,5,0+a01/2>,0.25 matte_white }
object { cylinder <-4,-5,0>,<-4,5,0+a03/2>,0.25 matte_white }
object { cylinder <-3,-5,0>,<-3,5,0+a05/2>,0.25 matte_white }
object { cylinder <-2,-5,0>,<-2,5,0+a07/2>,0.25 matte_white }
object { cylinder <-1,-5,0>,<-1,5,0+a09/2>,0.25 matte_white }
object { cylinder <-0,-5,0>,< 0,5,0+a10/2>,0.25 matte_white }
object { cylinder < 1,-5,0>,< 1,5,0+a02/2>,0.25 matte_white }
object { cylinder < 2,-5,0>,< 2,5,0+a04/2>,0.25 matte_white }
object { cylinder < 3,-5,0>,< 3,5,0+a06/2>,0.25 matte_white }
object { cylinder < 4,-5,0>,< 4,5,0+a08/2>,0.25 matte_white }
object { cylinder < 5,-5,0>,< 5,5,0+a11/2>,0.25 matte_white }
// our wall (a big disc, the edges don't show)
object {disc <0,0,15>,<0,0,1>,10 matte white }
// prepare a texture to reflect the scene on spheres
define mirror2
texture {
   surface {
      ambient white, -0.5
      diffuse white, 0.5
      specular O
      reflection white*2, 2
      }
   }
// make alternating growing and shrinking functions
// at 3 times the normal loop rate. Kissing spheres;
// squeeze1+squeeze2 = 0.5, the sphere spacing
```

```
continued from previous page
define squeeze1 0.25 + 0.05*sin(6*pi*index)
define squeeze2 0.25 - 0.05*sin(6*pi*index)
// 11 mirrored spheres to hide the spot_light overlap and
// show us what's going on behind the camera.
// "index"ing the x position variable makes the spheres
// travel right.
object {sphere <index-2.5,0,3>,squeeze1 mirror2 }
object {sphere <index-2.0,0,3>,squeeze2 mirror2 }
object {sphere <index-1.5,0,3>,squeeze1 mirror2 }
object {sphere <index-1.0,0,3>,squeeze2 mirror2 }
object {sphere <index-0.5,0,3>,squeeze1 mirror2 }
object {sphere <index+0.0,0,3>,squeeze2 mirror2 }
object {sphere <index+0.5,0,3>,squeeze1 mirror2 }
object {sphere <index+1.0,0,3>,squeeze2 mirror2 }
object {sphere <index+1.5,0,3>,squeeze1 mirror2 }
object {sphere <index+2.0,0,3>,squeeze2 mirror2 }
object {sphere <index+2.5,0,3>,squeeze1 mirror2 }
```

Steps

Here are the steps you need to follow to render and view this animation:

- 1. Change to the PLY\CHAPTER3\BARCODE directory.
- 2. Type PR BARCODE.
- 3. After 60 images render, type DTA barc* /0barc/s3/c4.
- 4. View the flic with AAPLAYHI.

How It Works

The vectors *c01-c11* define 11 different colors in RGB (red/green/blue) color specifications. They're mainly saturated colors (rather than pastels), needed to make distinct bright bands on the wall.

The variables *a01-a11* are 11 factors equally spaced along the length of a sine wave. The variable *amp* sets the gain so that these variables go from -2 to 2. We'll use these to shift the positions of the spotlights back and forth.

The variables b01-b11 are derived from a01-a11, but range from 0 to 2 to control the intensity of the lights.

Two rows of spotlights are defined, but rather than sequentially applying our position and light intensity factors to them, we scramble the variables to prevent the lights from coming on in waves.

The array of cylinders to intersect our lights is positioned along the plane where z = 0, and these cylinders are shifted plus or minus one unit about this plane during the animation.

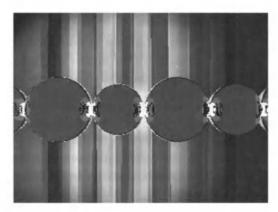


Figure 3-2 Colored bands with oscillating kissing spheres

A wall for the lights and shadows to fall on is created using a disc. A line of kissing mirrored spheres is generated to reflect the cylinders behind the camera, and a 60 frame animation is created. A sample of the output is shown in Figure 3-2.

Variations

You might try substituting some evolving geometric shape for the spheres, flex the disk or wrinkle it periodically, or place the lights on a conveyor belt like a treadmill and make all the colors and shapes warp in a more complex twisted fashion. This is one of those animations where you can just keep shoveling it in. Adding a collection of spiraling, orbital transparent spheres would be fun, because the colored bands would bend and blend inside them, but be aware that refraction takes much more time to render.



3.2 How do I...

Modify the texture of an object synchronized with the camera motion?

You'll find the code for this in: PLY\CHAPTER3\ROCKY

Problem

A viewpoint swaying back and forth in the trailing edges of a comet, spewing multihued gases would involve quite an elaborate particle systems animation and some fairly novel 3-D texturing scheme for the particles. But you can easily get a similar effect by viewing a shifting texture mapped on a sphere with a wide angle lens.

Technique

This animation is not difficult to produce, but the effects are difficult to watch. It's the insides of a sphere with a noise surface texture mapped on it, and we shift the phase of the color map and vary its turbulence as we rock it back and forth. Dramamine is highly recommended.

Noise Surface Textures

A few words on noise surfaces textures are appropriate at this point. Examine the rainbow texture in the following Polyray data file RAINBOW.PI.

```
// RAINBOW.PI
viewpoint {
   from <10,10,-10>
         <0, 1, 0>
   up
         <0,-3,0>
   at
   angle 40
   resolution 320,240
   aspect 1.333
light <1,1,1>,<-20,10,0>
light <1,1,1>,< 20,10,0>
background midnightblue
define rainbow
texture {
   noise surface {
      position_fn 5
      lookup_fn 1
      octaves 1
      turbulence 2
      ambient 0.2
      diffuse 0.8
      color_map(
      [0.00, 0.16, red, orange ]
      [0.16, 0.33, orange, yellow ]
      [0.33, 0.48, yellow, green ]
      [0.48, 0.67, green, blue ]
      [0.67, 0.83, blue, violet ]
      [0.83, 1.00, violet, black ])
   }
}
object {
   polygon 4, <-6,-1, -6>, <-6,-1, 6>,
              < 6,-1, 6>, < 6,-1, -6>
   rainbow
   }
```

The noise surface function generates a number between 0 and 1 for every point in 3-D space. These aren't exactly regular, but they're not completely random either. They vary slowly and smoothly over short distances, but appear random over longer ones. The mapping of these variations on an object and the way they vary are controlled by built-in Polyray position_fn and lookup_fn functions. You select a number for the effect you want and Polyray handles the details. The values 5 and 1, used here for position_fn and lookup_fn, specify varying the color map according to the distance around the y axis using a sawtooth function. Linear, spherical, and sine variations are also available. Check out the text listing just above the color_phase texture defined in ROCK1.PI (following) or look in the Polyray documentation for more details. Octaves and turbulence control the rate of change of the colors.

The *color_map* gives the relationship between the value returned by the noise function and its color. The first number in the color map specifies the start of the zone, the second the end, and the two colors after that the colors at both ends of a range. Colors in between are interpolated from the values at both ends. A value of 0.0 returns red, 0.08 (the midpoint of the first range) gives a color halfway between red and orange, 0.33 returns yellow and so on throughout the rainbow we've defined.

The shifting colors we're after are achieved by phasing the intensities of colors in a simple (but lengthy) *color_map*. The intensities of the 30 colors we create (each phased 12° apart for a total of 360°) vary sinusoidally across this *color_map* (see Figure 3-3). Holding red fixed, green is shifted left so that after 30 frames it's back where it started. Blue is shifted the same way, only at twice the speed of green. Somewhere around frame 23, the intensities of all three colors line up, which produces a black and white color map. The

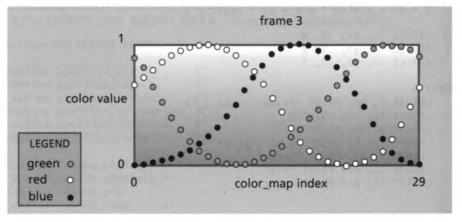


Figure 3-3 Phased intensities of red, green, and blue for 30 colors

rest of the time, we get a shifting rainbow. The code is lengthy and repetitive, so some QuickBasic code is written to automate the process and display how it works at the same time.

Writing The Program

The following two QuickBasic programs (COLORMAP.BAS and SHIFTY.BAS) write out the *color_map* and shows what's going on with the shifting intensities:

```
' COLORMAP.BAS
SCREEN 12
WINDOW (-16, -2)-(46, 2)
OPEN "colormap.inc" FOR OUTPUT AS #1
pi = 3.14159
rad = pi / 180
phz = 1
FOR phz = 0 TO 360 STEP 12
FOR n = 0 TO 30
  x = n * 12 + 6
   acount$ = "a" + RIGHT$("00" + LTRIM$(STR$(n)), 2)
   bcount = "b" + RIGHT (00" + LTRIM (STR (n)), 2)
   ccount$ = "c" + RIGHT$("00" + LTRIM$(STR$(n)), 2)
  IF phz=0 THEN
  PRINT #1, USING "define \ \ (1 + sin(### * rad)) / 2"; acount$, x
  PRINT #1, USING "define \ \ (1 + cos((phz + ###) * rad)) / 2"; bcount$, x
  PRINT #1, USING "define \ \ (1 - sin((2*phz + ###) * rad)) / 2"; ccount$, x
  PRINT #1,
   END IF
   r = (1 + SIN(x * rad)) / 2
   g = (1 + COS((phz + x) * rad)) / 2
   B = (1 - SIN((2 * phz + x) * rad)) / 2
   CIRCLE (n, r), .2, 4
   CIRCLE (n, g), .2, 2
   CIRCLE (n, B), .2, 1
NEXT n
   LOCATE 7, 37: PRINT "frame "; phz / 12
   LOCATE 16, 19: PRINT 0
   LOCATE 8, 19: PRINT 1
   LOCATE 12, 9: PRINT "color value"
   LOCATE 9, 61: PRINT "green"
   LOCATE 11, 61: PRINT "red"
   LOCATE 15, 61: PRINT "blue"
   LINE (0, 0)-(30, 1), 7, B
```

```
FOR x = 1 TO 29
      LINE (x, 0)-(x, .03)
   NEXT x
   LOCATE 17, 20: PRINT O
   LOCATE 17, 59: PRINT 29
   LOCATE 17, 33: PRINT "color_map index"
   LOCATE 22, 20: PRINT "Press any key to display the next colormap."
   DO WHILE INKEY$ = "": LOOP
  CLS
NEXT phz
  LOCATE 22, 23: PRINT "colormap written to colormap.inc."
CLOSE #1
             It creates the following listing in the file "COLORMAP.INC":
define a00 (1 + sin( 6 * rad)) / 2
define b00 (1 + cos((phz + 6) * rad)) / 2
define c00 (1 - sin((2*phz + 6) * rad)) / 2
define a01 (1 + sin( 18 * rad)) / 2
define b01 (1 + cos((phz + 18) * rad)) / 2
define c01 (1 - sin((2*phz + 18) * rad)) / 2
define a02 (1 + sin(30 * rad)) / 2
define b02 (1 + cos((phz + 30) * rad)) / 2
define c02 (1 - sin((2*phz + 30) * rad)) / 2
```

SHIFTY.BAS generates the lengthy *color_map* listing for us. This is an example of prepackaged code, where an editor macro is used to surround lengthy text listings with PRINT statements. When combined with programgenerated data, this technique produces ready-to-render Polyray code fragments. Your editor needs macro recording capabilities for you to use this effectively. The original code contained all those lines that start with "PRINT #2" in SHIFTY.BAS. SHIFTY.BAS is shown in the following listing.

```
OPEN "text1.inc" FOR OUTPUT AS #2
```

```
0"
PRINT #2, "define position_plain
PRINT #2, "define position_objectx
                                       1"
PRINT #2, "define position worldx
                                       2"
PRINT #2, "define position_cylindrical 3"
PRINT #2, "define position_spherical
                                       4"
PRINT #2, "define position radial
                                       5"
PRINT #2,
PRINT #2, "define lookup_plain
                                  0"
PRINT #2, "define lookup_sawtooth 1"
PRINT #2, "define lookup_sin
                                  2"
PRINT #2, "define lookup_ramp
                                  3"
PRINT #2,
PRINT #2, "define color_phase"
```

```
continued from previous page
PRINT #2, " noise surface {"

PRINT #2, " color white"

PRINT #2, " position_fn position_spherical"

PRINT #2, " lookup_fn lookup_ramp"

PRINT #2, " octaves 1"

PRINT #2, " turbulence 1.5+0.5*sin(index*rad*frame)"

PRINT #2, " ambient 0.2"

PRINT #2, " diffuse 0.8"

PRINT #2, " specular 0.3"

PRINT #2, " microfacet Reitz 5"

PRINT #2, " color_map("

FOR n = 0 TO 29
  FOR n = 0 TO 29
      n1 = n
      n2 = n + 1
      IF n2 = 30 THEN n2 = 0
      startvalue = n / 30
      endvalue = (n + 1) / 30
      a1$ = "a" + RIGHT$("00" + LTRIM$(STR$(n1)), 2)
      b1$ = "b" + RIGHT$("00" + LTRIM$(STR$(n1)), 2)
      c1$ = "c" + RIGHT$("00" + LTRIM$(STR$(n1)), 2)
      a2$ = "a" + RIGHT$("00" + LTRIM$(STR$(n2)), 2)
      b2$ = "b" + RIGHT$("00" + LTRIM$(STR$(n2)), 2)
      c2$ = "c" + RIGHT$("00" + LTRIM$(STR$(n2)), 2)
      PRINT #2, USING " [#.###, #.###, <\ \, \ \, \ \, \ \, \ \> ]"; ←
  startvalue, endvalue, a1$, b1$, c1$, a2$, b2$, c2$
  NEXT n
  PRINT #2, "
  PRINT #2, " }"
  PRINT #2, " scale <100,100,100>"
  PRINT #2, "}"
  CLOSE #2
```

Along with the color, we'll vary the turbulence and position of the texture as well. A wideangle camera from the interior of the sphere gives us a nice tunnel effect. Combining the previous listings with some additional details like lights and viewpoints gives us the following Polyray data file, ROCK1.PI.

```
// ROCK1.PI - Rocking color weirdness
start_frame 0
end_frame 29
total_frames 30
outfile "roc"
define pi 3.1415927
define rad pi/180
```

```
define phz 360*frame/total_frames
// lights
light <0.5, 0.5, 0.5>, < 180, 150, -150>
light <0.5, 0.5, 0.5>, < 0, 100, -15>
light <0.5, 0.5, 0.5>, < 0, 0, 0>
// camera
viewpoint {
  from <300,200,-250>
  at <0.0.0>
  up <0,1,0>
   angle 120
  aspect 1.433
  resolution 64,48
// action
define a00 (1 + sin( 6 * rad)) / 2
define b00 (1 + cos((phz + 6) * rad)) / 2
define c00 (1 - sin((2*phz + 6) * rad)) / 2
define a01 (1 + sin(18 * rad)) / 2
define b01 (1 + cos((phz + 18) * rad)) / 2
define c01 (1 - \sin((2*phz + 18) * rad)) / 2
define a 30 (1 + sin(366 * rad)) / 2
define b30 (1 + cos((phz + 366) * rad)) / 2
define c30 (1 - sin((2*phz + 366) * rad)) / 2
                            0
define position plain
define position_objectx
                            1
define position_worldx
                            2
define position cylindrical 3
define position_spherical
                            4
define position_radial
define lookup_plain
                       0
define lookup_sawtooth 1
define lookup_sin
                       2
                       3
define lookup_ramp
define color_phase
texture {
  noise surface {
      color white
      position_fn position_spherical
      lookup_fn lookup_ramp
      octaves 1
      turbulence 1.5+0.5*sin(phz*rad)
      ambient 0.2
```

```
continued from previous page
      diffuse 0.8
      specular 0.3
      microfacet Reitz 5
      color map(
      [0.000, 0.033, <a00, b00, c00>, <a01, b01, c01> ]
      [0.033, 0.067, <a01, b01, c01>, <a02, b02, c02> ]
      [0.067, 0.100, <a02, b02, c02>, <a03, b03, c03> ]
      [0.933, 0.967, <a28, b28, c28>, <a29, b29, c29> ]
      [0.967, 1.000, <a29, b29, c29>, <a00, b00, c00> ])
   scale <100,100,100>
}
define tx 100*sin(phz*rad)
define ty 50*sin(phz*rad)
define tz 100*sin(phz*rad)
// Create a volume
object {
   sphere <0,0,0>, 540
   color_phase
   translate <tx,ty,tz>
```

ROCK1.PI was generated manually by pasting together the outputs of COLORMAP.BAS and SHIFTY.BAS, creating a main file which defined the camera, light, output files, and objects. The same thing could have been done including those outputs in the main file by using "include" commands, as shown in the following example, ROCK2.PI.

```
// ROCK2.PI - Rocking color weirdness, done with include files
start_frame 0
end_frame 29
total_frames 30
outfile "roc"

define pi 3.1415927
define rad pi/180

define phz 360*frame/total_frames

// Lights
light <0.5, 0.5, 0.5>, < 180, 150, -150>
light <0.5, 0.5, 0.5>, < 0, 100, -15>
light <0.5, 0.5, 0.5>, < 0, 0, 0>

// Camera
```

```
viewpoint {
   from <300,200,-250>
   at <0,0,0>
   up <0,1,0>
   angle 120
   aspect 1.433
   resolution 320,200
// Action
include "colormap.inc"
include "text1.inc"
define tx 100*sin(phz*rad)
define ty 50*sin(phz*rad)
define tz 100*sin(phz*rad)
// Create a volume
object {
   sphere <0,0,0>, 540
   color_phase
   translate <tx,ty,tz>
```

Both ROCK1.PI and ROCK2.PI generate 30 frames that resemble Figure 3-4, except they sway.

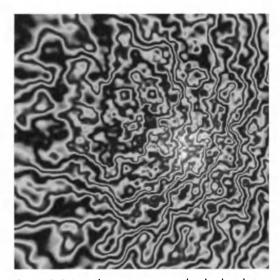


Figure 3-4 A wiggly streaming texture that develops during our animation

How It Works

This animation works by shifting the phase of the red, green, and blue portions of a series of colors to produce shifting periodic color bands. The motivator needs to be a phase angle varying from 0 to 360° over the course of the total frame count, which in this case is 30. The variable phz, defined as

```
define phz 360*frame/total_frames
```

fills the bill. The variables *a00-a29*, *b00-b29* and *c00-c29* specify the red, green, and blue portions of our color map. Note that in the script

```
define a00 (1 + sin( 6 * rad)) / 2
define b00 (1 + cos((phz + 6) * rad)) / 2
define c00 (1 - sin((2*phz + 6) * rad)) / 2
```

a00—the red intensity—doesn't depend on phz, making it fixed throughout the animation. Green varies by phz, and blue by twice that (2*phz). We generate 30 colors that span the spectrum by adding a 12° offset to each successive color.

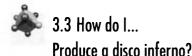
We used defines to convert meaningless numbers into a text description of the available mapping functions. While the numbers would have worked just as well, the text is far more informative. A noise surface is defined, where we vary the turbulence between 1 and 2 during the animation to stir things up a bit:

```
turbulence 1.5+0.5*sin(phz*rad)
```

The long *color_map* listing—which uses the colors defined in the previous section—is compiled and scaled up 100 times until it is the right size for the 540-unit sphere we're mapping it onto. Phased translations for the texture as *tx*, *ty*, *tz* are generated, rocking the texture back and forth sinusoidally on the sphere. Once the sphere is completed, map the texture onto it, and shift that texture around.

Comments

There are an endless number of variations on this simple animation. Experiment with the various texture mapping functions and turbulence values, shift the colors of the lights, vary the bumpiness of the texture or create a whole series of spheres and fly through them.



You'll find the code for this in: PLY\CHAPTER3\NTCLUB

Problem

Lights! Tables! Loud music! Alien cubes from the 4th dimension! No, wait, that wasn't supposed to be there... Of course it was! This is night life! Dance clubs are designed to overload the sensory inputs of both the eyes and the ears, with little regard for style or taste. The louder the music, the brighter the colors, the greater the number of frantically panning spotlights, the better. This sounds like a job for an amazing color_map (correct) and several hundred panning spot_light (wrong!). Rendering time increases dramatically the more lights you use, so keeping it down to some reasonably small number, say one, would be a better idea.

Technique

Rather than lots of lights, we'll use a single shaped light source known as a textured_light that allows us to functionally describe how light surrounds its source, and then tumble the source to splash the interior of a room with shifting light patterns. We'll place this source inside an object that's defined to be the same shape as the distribution of color from the source. It's a cubical, tooth-like, toaster shaped object used in an article by Don Mitchell and Pat Hanrahan on the illumination of curved reflectors for the 1992 SIGGRAPH Proceedings. We'll use a moving color_map to paint the walls, making them appear to crawl right out of the floor, looking like badly overexposed tie-dyed T-shirts. This animation uses way too much color. Tables, each with their own spotlight, and an exit door (to allow any remnants of good taste to leave) complete the image.

Steps

We use a box object for our room that's 10 units by 10 units with a ceiling 5 units up. This must either be a metric bar or more a club house than a night club. We generate a fairly intense *color_map* called *ripple_rainbow_texture* that will be mapped onto the entire box. A checkered floor is added. Circular tables are made with a disc, a large cylinder in the form of a band for the table edges, and a smaller diameter pillar to support them both. Six copies are

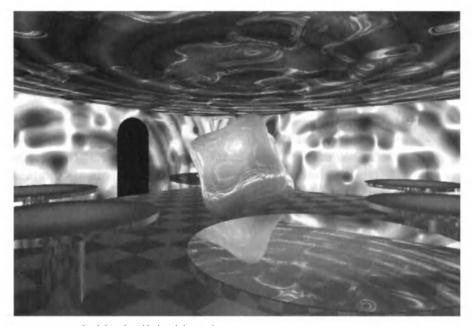


Figure 3-5 Night club with gold plated disco cube

scattered around the room, and the last one, closest to the camera, is given a reflective top. It's a bit too busy if all of them have shiny tops. Each table is given its own *spot_light* for an intimate mood, and the cube in the center of the room is surrounded by four additional spotslights (see Figure 3-5). Ah yes, the cube. It's not obvious at first glance that the equation

$$x^4 + y^4 + z^4 - x^2 - y^2 - z^2$$

ends up looking like the round-edged disco cube in Figure 3-5, but it does. We use it to define both an object and a light source, stick this light source inside the cube, and let the light escape by turning off the cube's shadow using shading flags. The cube tumbles, imaginary music throbs, and little flecks of phosphor peel off the insides of your VGA monitor in a desperate attempt to faithfully display the oversaturated colors (just kidding). This is handled by the following listing, NTCLUB.PI.

```
// NTCLUB.PI
// Polyray input file: Jeff Bowermaster
start_frame 0
end_frame 89
total_frames 90
outfile ntclb
```

```
define frame_normal frame/total_frames
define pi 3.14159
define rad pi/180
// set up the camera
viewpoint {
  from <-7.25, 2.5, -5.25>
  at <0,2.5,0>
  up <0,1,0>
  angle 60
  resolution 320,200
  aspect 1.43
  }
include "\ply\colors.inc"
define ang frame_normal*2*pi
// the tumbling motion from Chapter 2.6
define factor 90
define xr factor * (2 * SIN(ang) + SIN(3 * ang) / 3)
define yr factor * (2 * COS(ang) - COS(3 * ang) / 3)
define zr factor * (0.9*COS(2 * ang))
//light <0.5, 0.5, 0.5>, <-4, 4.9, -4>
// pillow/tooth object from the back cover of the 1992 Siggraph
// proceedings: (x^4 + y^4 + z^4 - x^2 - y^2 - z^2)
// make it both the object and the light source
textured light {
  color (x^4 + y^4 + z^4 - x^2 - y^2 - z^2)/3000
  rotate <xr,yr,zr>
  translate <0, 2, 0>
object {
  object {
      polynomial x^4 + y^4 + z^4 - x^2 - y^2 - z^2
      shading_flags 32 + 8 + 4 + 2 +1 // turn off the objects shadow
      root_solver Ferrari
      }
  rotate <xr,yr,zr>
   translate <0, 2, 0>
  reflective_coral
  }
// tons of color
define ripple_rainbow_texture
                                                                  continued on next page
```

CHAPTER THREE

```
continued from previous page
texture {
   noise surface {
      position_fn 1
      lookup_fn 3
      octaves 4
      turbulence 3*sin(ang)
      normal 1
      frequency 1
      ambient 0.1
      diffuse 0.5
      specular 0.6
      reflection 0.5
      microfacet Reitz 10
      color_map(
         [0.000, 0.122, <0.467, 0.075, 0.075>, <0.922, 0.000, 0.000>]
         [0.122, 0.296, <0.922, 0.000, 0.000>, <1.000, 1.000, 0.000>]
         [0.296, 0.383, <1.000, 1.000, 0.000>, <0.227, 0.796, 0.404>]
         [0.383, 0.626, <0.227, 0.796, 0.404>, <0.000, 0.000, 0.745>]
         [0.626, 0.809, <0.000, 0.000, 0.745>, <0.388, 0.180, 0.698>]
         [0.809, 0.887, <0.388, 0.180, 0.698>, <0.365, 0.141, 0.439>]
         [0.887, 0.948, <0.365, 0.141, 0.439>, <0.110, 0.110, 0.110>]
         [0.948, 1.001, <0.110, 0.110, 0.110>, <0.467, 0.075, 0.075>])
      translate <0.0, frame/10, 0.0 >
   }
// color for the doorway
define jet_black
texture {
   surface {
      ambient black, 0.0
      diffuse black, 0.0
      }
   }
// this is the room we are in
object { box <-10, 0, -10>, <10, 5, 10> ripple_rainbow_texture }
object { box < 2, 0, 10.1>, <4, 3.5, 9.9> jet_black } // door
object { disc <3, 3.5, 9.9>, <0, 0, 1 >, 1 jet_black }
                                                          // arch over door
define shiny_cornflower texture { shiny { color CornFlowerBlue } }
// A checker floor
object {
   polygon 4, <-10, 0.1, -10>, <-10.1, 0, 10>,
              < 10, 0.1, 10>, < 10.1, 0, -10>
   texture {
      checker shiny_coral, shiny_cornflower
      translate <0, -0.1, 0>
      scale <1, 1, 1>
      }
   }
```

```
// these are tables in the room
define table
  object {
      object { disc <0, 1.5, 0>, <0, 1.5, 0>, 1.5 shiny_red }
    + object { cylinder <0, 1.5, 0>, <0, 1.4, 0>, 1.5 reflective_red }
    + object { cylinder <0, 1.5, 0>, <0, 0.0, 0>, 0.3 reflective_red }
table { translate < 0, 0, -5 > }
table { translate <-5, 0, 0> }
table { translate <-4, 0,-4> }
table { translate < 4, 0,-4> }
table { translate < 4, 0, 4> }
table { translate <-4, 0, 4> }
// make one tabletop reflective
object {
  disc <0, 1.51, 0>, <0, 1.5, 0>, 1.5
  reflective_red
  translate <-4,0,-4>
}
// give each of the tables its own spot
spot_light white, < 0.4.9, -5>, < 0.0, -5>, 3, 5, 20
spot_light white, <-5,4.9, 0>, <-5,0, 0>, 3, 5, 20
spot_light white, <-4,4.9,-4>, <-4,0,-4>, 3, 5, 20
spot_light white, <-4,4.9, 4>, <-4,0, 4>, 3, 5, 20
spot_light white, < 4,4.9,-4>, < 4,0,-4>, 3, 5, 20
spot_light white, < 4,4.9, 4>, < 4,0, 4>, 3, 5, 20
// spotlight the tumbling shape in the center of the room
spot_light <0.7, 0.7, 0.7>, <-4,2.5, 0>, <0,2.5,0>, 3, 5, 20
spot_light <0.7, 0.7, 0.7>, < 4,2.5, 0>, <0,2.5,0>, 3, 5, 20
spot_light <0.7, 0.7, 0.7>, < 0,2.5,-4>, <0,2.5,0>, 3, 5, 20
spot_light <0.7, 0.7, 0.7>, < 0,2.5, 4>, <0,2.5,0>, 3, 5, 20
```

How It Works

The *ripple_rainbow_texture*, mapped onto the entire interior surface of our room, provides a simple means of simultaneously modifying both the walls and the ceiling with one simple translation statement in the texture:

```
translate <0.0, frame/10, 0.0 >
```

This texture moves up 0.1 units each frame, which makes the walls flow up from the floor and makes the ceiling boil, since it's the cross section of a turbulent 3-D texture. However, for linear translations of random textures, there is no smooth loop point. All we can do is minimize the complexity of the texture by making the turbulence factor go to zero at the loop point. This simplifies the texture down to a series of vertical bands on one wall and a

solid color on the other. It won't look too bad suddenly shifting 9 units (90 frames * 0.1 units per frame).

The focal point of the animation is the tumbling cube. The code in Section 2.6 is used for the motion. We spotlight it to show off how shiny it is. However, this is nothing compared to what happens when we place a textured light inside it and turn off its shadow to let the light out.

Shadow Flags

Frequently used variables are often grouped together as collections of bits in a binary number. For example, 4 is expressed as 00000100. A branch function in the rendering code would check to see if this bit was on or off, and execute some code accordingly. Single bit comparisons are very fast operations, so it makes sense to use them instead of longer integer comparisons in cases where the code executes a great number of times. The shading quality flags are just such variables. By specifying a series of bits, various rendering options can be turned on and off. The bit that determines whether an object casts shadows is specified by 16 (00010000). Leaving this value out and summing all the other values:

shading_flags 32 + 8 + 4 + 2 +1 // turn off the object's shadow

allows light from the inside of an object to pass through, although the object still appears to be solid. See the Polyray documentation for the other flags meanings.

For reasons we don't need to understand to exploit, the textured light source inside our tumbling cube is several times more efficient at illuminating the walls than it is the ceiling. The same effect could have been achieved using a darker texture on the ceiling, but this is an unexpected bonus. We don't need to deal with the ceiling separately to achieve the moody darkness that most clubs use to hide the duct work and the wiring for the lights and speakers.



3.4 How do I...

Generate a spline camera path through a tunnel?

You'll find the code for this in: PLY\CHAPTER3\TUNL

Problem

Spline paths are required for smooth, non-repetitive motions in and around objects. Webster's defines splines as pliable strips of wood or metal. They may be bent into various shapes and held that way with pegs. Computers

bend curves and use control points and tension values to achieve the same effect. Computer-generated spline paths are an ideal platform for graceful fly-through motions.

Many fairly thick books have been written on the topic of splines, and the math is neither friendly nor pleasant. Some of it can stun at 20 yards. OK, an exaggeration, 10 yards is more typical. Suffice it to say there are *lots* of ways to create splines and the topic has been covered in all its gory detail elsewhere. We're going to focus here on splines that change their direction smoothly as they pass *near* their control points (as opposed to *through* their control points). David Mason's SPPATH program (also called SP and covered in more detail in the Waite Group book *Making Movies on Your PC*) does a great job with the *through* types.

Through Versus Near—Pros and Cons

The advantage of the *through* spline types is that you know exactly where they'll be in relation to the control points. Their disadvantage is they tend to have a harder feel to them and are less forgiving when the control points are misplaced. They might jerk you around a bit.

Splines that, on the other hand, simply pass *near* their control points are generally smoother, but you might wander inadvertently through walls or floors during tight turns, even though your control points stay well clear of such objects. Their other disadvantage is that they throw away their first and last points. The concept of *near* relative to the next point is undefined when there is no next point. So in order to create a looping spline path from this type of spline, extra points must be added at the ends to overlap the control point sets.

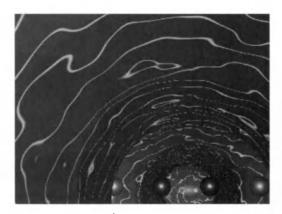


Figure 3-6 Our tunnel

Technique

This animation was inspired by the terrific tunnel fly-throughs of Chris Smotherman. A view of the tunnel we're building is shown in Figure 3-6.

The tunnel is composed of two perpendicular sets of three parallel cylindrical blobs, basically a square with opposite sides connected like cross hairs (see Figure 3-7). Control points (43 open circles) are scattered around the tunnel, and the QuickBasic program generates a smoothed path near these points (dotted line) as a series of 43 individual segments. The circles appearing on this dotted line are actually the endpoints of each individual segment that smoothly join together to form the path.

The program writes two files during the calculations. The file CUBIC.INC is the collection of the cubic coefficients for each segment, and SPLINE.3D is a file that can be viewed with Oscar Garcia's real time mouse controlled wire

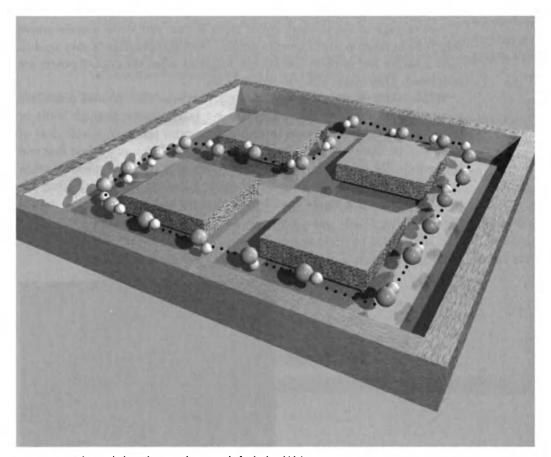


Figure 3-7 Spline path through a tunnel composed of cylindrical blobs

frame 3-D viewer, 3DV. The coefficient files allow us to create the individual camera positions (10 per segment) inside Polyray, rather than resorting to batched *include* files.

Steps

The first step is to place this model on a grid and decide where we want the path to go. The only way to generate the individual control points is to choose them manually. We'll modify certain portions of the path later programmatically, but for starters, you have to decide where the camera should go.

The following code (PATH.BAS) re-creates Figure 3-6 (in color this time) and writes CUBIC.INC (which contains the cubic coefficients for regenerating the spline sections within Polyray), and SPLINE.3D (the 3DV-viewable spline path).

```
' PATH.BAS - Spline Path Creation Program
i = 100
DIM x(i), y(i), z(i)
DIM red(16), green(16), blue(16)
'rainbow palette
DATA 0, 0, 0
DATA 32, 0, 0
DATA 42, 0, 0
DATA 58, 16, 0
DATA 63, 32, 0
DATA 58, 56, 0
DATA 16, 42, 0
DATA 0, 30, 36
DATA 0, 20, 40
DATA 0, 10, 48
DATA 0, 0, 63
DATA 20, 0, 53
DATA 23, 0, 29
DATA 19, 7, 17
DATA 50, 40, 45
DATA 63, 63, 63
SCREEN 12
WINDOW (-10, -7.5)-(10, 7.5)
FOR y = 1 TO 4
       FOR x = 1 TO 4
                colornum = x + ((y - 1) * 4) - 1
               READ red(colornum), green(colornum), blue(colornum)
               KOLOR = 65536 * blue(colornum) + 256 * green(colornum) +
red(colornum)
                                                                 continued on next page
```

CHAPTER THREE

```
continued from previous page
                PALETTE colornum, KOLOR
                COLOR colornum
       NEXT x
NEXT y
FOR y = -7 TO 7
       LINE (-8, y / 2)-(-7.55, y / 2 + .45), y + 8, BF
NEXT y
' left to right tunnels
LINE (-6, -.5)-(6, .5), B
LINE (-6, 5.5)-(6, 6.5), , B
LINE (-6, -6.5)-(6, -5.5), B
' up and down tunnels
LINE (-6.5, -6)-(-5.5, 6), B
LINE (-.5, -6)-(.5, 6), B
LINE (5.5, -6)-(6.5, 6), B
pi = 3.14159
rad = pi / 180
' the individual control points: graph paper time.
DATA -6,0.01,-1
DATA -6,0.01,0
DATA -5,0.01,0
DATA -4,0.01,0
DATA -3,0.01,0
DATA -2,0.01,0
DATA -1,0.01,0
DATA 0,0.01,0
DATA 0,0.01,1
DATA 0,0.01,2
DATA 0,0.01,3
DATA 0,0.01,4
DATA 0,0.01,5
DATA 0,0.01,6
DATA 1,0.01,6
DATA 2,0.01,6
DATA 3,0.01,6
DATA 4,0.01,6
DATA 5,0.01,6
DATA 6,0.01,6
DATA 6,0.01,4.75
DATA 6,0.01,3.25
DATA 6,0.01,1.50
DATA 6,0.01,-0.5
DATA 6,0.01,-2.25
DATA 6,0.01,-3.75
DATA 6,0.01,-5
```

```
DATA 6,0.01,-6
DATA 4.75,0.01,-6
DATA 3.25,0.01,-6
DATA 1.5,0.01,-6
DATA -0.5, 0.01, -6
DATA -2.25,0.01,-6
DATA -3.75, 0.01, -6
DATA -5,0.01,-6
DATA -6.0.01.-6
DATA -6,0.01,-5
DATA -6,0.01,-4
DATA -6.0.01.-3
DATA -6,0.01,-2
DATA -6,0.01,-1
DATA -6,0.01,0
DATA -5,0.01, 0
m = 43
n = 12
count = 493
wide = 3
pi = 3.14159
rad = pi / 180
amp = .5
' the original focus of the animation was the flight
' through the central section of the tunnel. The following code
' creates a wide spiral path through this section, under the control
' of some simple-to-modify scaling variables, to create a loop
' ranging from an elaborate roll to a flat 90 degree turn
FOR i = 1 TO m
   READ x(i), y(i), z(i)
   MAGx = amp / (EXP((x(i) / wide) ^ 2))
   MAGz = amp / (EXP((z(i) / wide) ^ 2))
   shift = 1
   IF x(i) \Leftrightarrow 0 THEN ' on approach
      xm = 0
      ym = MAGx * COS((x(i) + shift) * 90 * rad)
      zm = MAGx * SIN((x(i)) * 90 * rad)
   ELSE
                        ' after pass through
      xm = MAGz * SIN((z(i)) * 90 * rad)
      ym = MAGz * COS((z(i) + shift) * 90 * rad)
      zm = 0
  END IF
  x(i) = x(i) + xm
  y(i) = y(i) + ym
  z(i) = z(i) + zm
  CIRCLE (x(i), z(i)), .1, y(i) * 20 + 8
NEXT i
```

CHAPTER THREE

```
continued from previous page
OPEN "cubic.inc" FOR OUTPUT AS #1
OPEN "spline.3d" FOR OUTPUT AS #2
PRINT #2, count
c = -8
FOR i = 2 TO m - 1
       ' MAGIC SPLINE FITTING CODE
       xA = x(i - 1): xB = x(i): xC = x(i + 1): xD = x(i + 2)
       yA = y(i - 1): yB = y(i): yC = y(i + 1): yD = y(i + 2)
       zA = z(i - 1): zB = z(i): zC = z(i + 1): zD = z(i + 2)
       a3 = (-xA + 3! * (xB - xC) + xD) / 6!
       a2 = (xA - 2! * xB + xC) / 2!
       a1 = (xC - xA) / 2!
       a0 = (xA + 4! * xB + xC) / 6!
       b3 = (-yA + 3! * (yB - yC) + yD) / 6!
       b2 = (yA - 2! * yB + yC) / 2!
       b1 = (yC - yA) / 2!
       b0 = (yA + 4! * yB + yC) / 6!
       c3 = (-zA + 3! * (zB - zC) + zD) / 6!
       c2 = (zA - 2! * zB + zC) / 2!
       c1 = (zC - zA) / 2!
       c0 = (zA + 4! * zB + zC) / 6!
       PRINT #1, USING "< ##.###, ##.###, ##.### >"; a3, b3, c3
       PRINT #1, USING "< ##.####, ##.####, ##.### >"; a2, b2, c2
       PRINT #1, USING "< ##.####, ##.####, ##.### >"; a1, b1, c1
       PRINT #1, USING "< ##.####, ##.####, ##.#### >"; a0, b0, c0
       PRINT #1,
       w = 2
       FOR j = first TO n
               t = j / n
              x = ((a3 * t + a2) * t + a1) * t + a0
              y = ((b3 * t + b2) * t + b1) * t + b0
              z = ((c3 * t + c2) * t + c1) * t + c0
              PRINT #2, USING "##.##### "; x, y, z
               IF j = first THEN
                      PSET (x, z), y * 20 + 8
                      CIRCLE (x, z), .05, 2
              ELSE
                      PSET (x, z), y * 20 + 8
              END IF
       NEXT j
       first = 1
NEXT i
CLOSE #1
```

How It Works

The trick now is to use the cubic coefficients in CUBIC.INC to generate the camera motion through the tunnel. We write a program that reads the coefficients out of the CUBIC.INC data file and generates 39 separate Polyray data files that are called sequentially. This is not an elegant solution, but the extra points you get for style don't count much once an animation is completed.

The file CUBIC.INC looks like this:

```
< -0.1667, -0.0155, 0.1822 >
< 0.5000, 0.0155, -0.5155 >
< 0.5000, 0.0155, 0.4845 >
< -5.8333, 0.0152, -0.1718 >
< 0.0000, -0.0151, 0.0151 >
< 0.0000, -0.0311, 0.0311 >
< 1.0000, -0.0000, 0.0000 >
< -5.0000, 0.0307, -0.0207 >
< 0.0000, 0.0868, -0.0868 >
< 0.0000, -0.0764, 0.0764 >
< 1.0000, -0.1075, 0.1075 >
< -4.0000, -0.0155, 0.0255 >
< 0.0000, -0.0174, 0.0174 >
< 0.0000, 0.1839, -0.1839 >
< 1.0000, 0.0000, -0.0000 >
< -3.0000, -0.1126, 0.1226 >
```

The QuickBasic program sequentially read CUBEREAD.BAS has been created to two sets of spline coefficients—one for the current location, the other for the location where it's pointed. The camera will always be pointed at corresponding locations on the next segment down its path. CUBEREAD. BAS appears in the following listing.

```
' CUBEREAD.BAS
FOR x = 0 T0 39
  prefix$ = "\ply\dat\blobs\tunl"
  count$ = RIGHT$("00" + LTRIM$(STR$(x)), 2)
  filename$ = prefix$ + count$ + ".pi"
```

```
continued from previous page
   OPEN filename$ FOR OUTPUT AS #1
   PRINT #1, "start_frame "; x * 12
   PRINT #1, "end_frame"; (x + 1) * 12 - 1
   PRINT #1, "total_frames 12"
   PRINT #1,
   PRINT #1, "outfile "; CHR$(34); "tunl"; CHR$(34)
   OPEN "cubic.inc" FOR INPUT AS #2
   'pre-read
     FOR y = 1 T0 x * 5
       LINE INPUT #2, a$
     NEXT y
     LINE INPUT #2, a$
     PRINT #1, "define a3 "; a$
     LINE INPUT #2, a$
     PRINT #1, "define a2 "; a$
     LINE INPUT #2, a$
     PRINT #1, "define a1 "; a$
     LINE INPUT #2, a$
     PRINT #1, "define a0 "; a$
     LINE INPUT #2, a$
     PRINT #1,
     LINE INPUT #2, a$
     PRINT #1, "define b3 "; a$
     LINE INPUT #2, a$
     PRINT #1, "define b2 "; a$
     LINE INPUT #2, a$
     PRINT #1, "define b1 "; a$
     LINE INPUT #2, a$
     PRINT #1, "define b0 "; a$
     LINE INPUT #2, a$
     PRINT #1,
   CLOSE #2
   OPEN "tunl" FOR INPUT AS #2
   DO WHILE NOT EOF(2)
      LINE INPUT #2, a$
      PRINT #1, a$
   L00P
   CLOSE #2
   CLOSE #1
NEXT x
```

The code that converts the cubic coefficients into camera positions, as well as the lights, objects, and textures, is read in from the file TUNL. An example of the output for this program follows. This is the ninth spline segment (TUNL09.PI) renders frames 108 to 119 and is written as follows:

```
(TUNLO9.PI)
start_frame 108
end_frame 119
total_frames 12
outfile "tunl"
define a3 < -0.0868, 0.0868, 0.0000 >
define a2 < 0.1839, -0.1839, 0.0000 >
define a1 < -0.0000, 0.0000, 1.0000 >
define a0 < -0.1226, 0.1326, 3.0000 >
define b3 < 0.0151, -0.0151, 0.0000 >
define b2 < -0.0764, 0.0764, 0.0000 >
define b1 < 0.1075, -0.1075, 1.0000 >
define b0 < -0.0255, 0.0355, 4.0000 >
define t (frame - start_frame) / total_frames // goes from 0 -> almost 1
// now we do the splined position for a given frame
// then the position can be calculated with:
define pos (((a3 * t + a2) * t + a1) * t + a0)
define at1 (((b3 * t + b2) * t + b1) * t + b0)
viewpoint {
  from pos
  up
        <0, 1, 0>
  at
         at1
  angle 30
  resolution 160,120
  aspect 1.333
light <1,1,1>,pos
define pi 3.14159
define rad pi/180
// red blinking light
light <1+sin(frame*rad*12),0,0>,<0, 3, 0>
background midnightblue
include "\ply\colors.inc"
define blue_buzz
texture {
  noise surface {
     color white
     position fn 4
     lookup_fn 1
     octaves 4
     turbulence 3
```

```
continued from previous page
      ambient 0.2
      diffuse 0.6
      specular 0.3
      microfacet Reitz 5
      color map(
        [0.000, 0.600, <0.100, 0.000, 0.740>, <0.100, 0.000, 0.740>]
        [0.600, 0.860, <0.100,0.000,0.740>, <0.400,0.700,1.000>]
        [0.860, 0.920, <0.400,0.700,1.000>, <3.000,1.500,0.000>]
        [0.920, 0.970, <3.000,3.000,0.000>, <3.000,3.000,0.000>]
        [0.970, 1.000, <3.000,1.500,0.100>, <0.100,0.000,0.740>])
   }
   scale <0.5,0.5,0.5>
}
// the tunnel we're in
object {
   blob 0.50:
      cylinder <-6, 0, -6>, < 6, 0, -6>, 1, 1.0,
      cylinder <-6, 0, 0>, < 6, 0, 0>, 1, 1.0,
      cylinder <-6, 0, 6>, < 6, 0, 6>, 1, 1.0,
      cylinder <-6, 0, -6>, <-6, 0, 6>, 1, 1.0,
      cylinder < 0, 0, -6>, < 0, 0, 6>, 1, 1.0,
      cylinder < 0, 0, 6>, < 0, 6, 6>, 1, 1.0,
      cylinder < 6, 0, -6>, < 6, 0, 6>, 1, 1.0
      blue_buzz
}
// some obstacles
object {
   sphere <0,0.1,3.5-frame/20>, 0.05
   shiny_red
   }
   sphere <0,0.1,4.0-frame/20>, 0.05
   shiny_blue
   }
object {
   sphere <0,0.1,4.5-frame/20>, 0.05
   shiny_blue
   }
object {
   sphere <0,0.1,5.0-frame/20>, 0.05
   shiny_red
   }
   sphere <0.0.1,5.5-frame/20>, 0.05
   shiny_red
```

```
bobject {
    sphere <0,0.1,6.0-frame/20>, 0.05
    shiny_blue
}

object {
    sphere <0,0.1,6.5-frame/20>, 0.05
    shiny_red
}

object {
    sphere <0,0.1,7.0-frame/20>, 0.05
    shiny_blue
}

object {
    sphere <0,0.1,7.5-frame/20>, 0.05
    shiny_red
}
```

All that's left is to generate a batch file to call all 39 Polyray data files. The batch file that's normally used to call Polyray, PR.BAT, reads:

```
\ply\polyray %1.pi -o %1.tga %2
```

To call a batch file from another batch file, just in case you've never tried this, requires you CALL it. A batch file RUNEM.BAT reads:

```
CALL pr TUNLO1
CALL pr TUNLO2
CALL pr TUNLO4
CALL pr TUNLO5
CALL pr TUNLO6
CALL pr TUNLO7
CALL pr TUNLO7
CALL pr TUNLO8
CALL pr TUNLO9
CALL pr TUNLO9
CALL pr TUNLO10
```

Comments

Down one stretch (spline points 29-37), the linear motion was replaced with some accelerated motion. The x variable, rather than indexing by 1, indexed as 6, 4.75, 3.25, 1.5, -0.5, -2.25, -3.75 and -5.0. These coefficients were generated using the following code (STRETCH.BAS).

```
' STRETCH.BAS
SCREEN 12
WINDOW (-6.4, -4.8)-(6.4, 4.8)
```

```
continued from previous page
'OPEN "accel" FOR OUTPUT AS #1
FOR x = -6 TO 6 STEP 1
   CIRCLE (x, 0), .25, 4
NEXT x
FOR a = .20 \text{ TO } .30 \text{ STEP } .001
LOCATE 1, 1: PRINT a
x = -7
v = 1
DO WHILE x < 6.5
   CIRCLE (x, 1), .15, 2
 ' PRINT #1, x
   x = x + v
   LOCATE 2, 1: PRINT v
 ' accelerate halfway, then decelerate
   IF x < 0 THEN
      v = v + a
  ELSE
     v = v - a
  END IF
L00P
LOCATE 22, 10: PRINT "PRESS A DEY TO STEP TO THE NEXT VALUE, CTRL-BREAK TO END"
DO WHILE INKEY$ = "": LOOP
LINE (-6.5, .5)-(6.5, 1.5), 0, BF
NEXT a
'CLOSE #1
```

The desired goal was to generate an acceleration/deceleration curve that smoothly meshed with the existing linear spline curve at the endpoints. A loop was set up where the acceleration (a) varied over a small range, and the resulting points were displayed above the existing linear ones. When the endpoints lined up, the values for the positions of the acceleration/deceleration points could be used to write new spline control points by changing (a) in the code and enabling the PRINT #1 code.

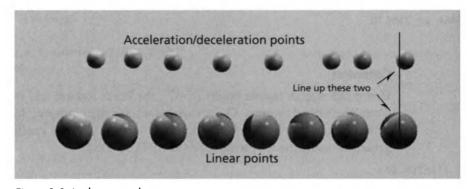


Figure 3-8 Acceleration meshing program output



🏅 3.5 How do I...

Simulate motion down an endless tunnel with a simple model and a few frames?

You'll find the code for this in: PLY\CHAPTER3\FAKE

Problem

Let's produce an animation where we run down a very long hallway very fast. Long tunnels that stretch off into vanishing points containing details like doorways, adjoining tunnels, and ceiling lights must contain enormous numbers of graphics primitives. The foreground details have to be repeated until the end of the tunnel is just a guess, a few pixels across. Large numbers of objects can take an awfully long time to render.

A solution that speeds up rendering time and simplifies the modeling uses strategic placement of mirrors both in front of and behind the camera, and allows infinite reflections to create the illusion of a tunnel that stretches on forever (Figure 3-9).

Technique

Barber shops used to have parallel mirrors on their walls to allow you to see the back of your head while you had your hair cut. The reflections stretched on forever, but your face usually got in the way, and impurities and imperfections in the glass limited the range. Fortunately, ray tracers have



Figure 3-9 Fake endless tunnel

perfect mirrors and you don't have to worry about the camera getting in the way because it's invisible. Objects placed between two parallel mirrors will repeat forever, giving the illusion of a world filled with them, extending off into infinity. All we'll need is a short piece of hallway that has exactly the same geometry from one end to the center as it does from the center to the far end. We place mirrors at both ends, and start with the camera just in front of one mirror. We move the camera down the corridor until we're almost touching the other mirror at the far end. With the correct choice of camera steps, the animation will repeat perfectly, and continuous motion down a corridor will be achieved.

Steps

We're after a tunnel that extends forever in front of the camera, or at least far enough so you can't see the end. For contrast, let's start off doing it the hard way, with a physical model that repeats the same tunnel geometry 20 times. In the following listing (REAL.PI), we construct walls from thin boxes (like drywall), map repeating patterns onto them, then cruise off down the hall the distance of one repeating unit in ten frames:

```
// REAL.PI - Image-mapped Endless Tunnel with a Real Tunnel
     we'll use three targa files as image maps (any will do):
//
     the floor - floor.tga
//
     the walls - walls.tga
//
     the ceiling - ceiling.tga
//
     they can be the ones supplied or anything you wish.
start_frame 0
end frame 9
total_frames 10
outfile "real"
include "\ply\colors.inc"
// move the camera 5 units forward each frame
viewpoint {
   from <0,6,-10+frame*5>
   at <0,0,200+frame*5>
   up <0,1,0>
   angle 30
   resolution 320,200
   aspect 1.6
   }
```

background midnightblue

define dim 0.6

```
define disc_image1 image("floor.tga")
define the_floor
   texture {
      special surface {
         color planar_imagemap(disc_image1, P, 1)
         ambient 0.2
         diffuse 0.8
      }
      translate <-0.5, 0, -0.5>
      scale <1,1,1> -rotate <0,0,90>
   }
define disc_image2 image("walls.tga")
define the_walls
   texture {
      special surface {
         color planar_imagemap(disc_image2, P, 1)
         ambient 0.2
         diffuse 0.8
      }
      translate <-0.5, 0, -0.5>
      rotate <0.0.90>
      scale <1,1,1>
   }
define disc_image3 image("ceiling.tga")
define the ceiling
   texture {
      special surface {
         color planar_imagemap(disc_image2, P, 1)
         ambient 0.2
         diffuse 0.8
      translate <-0.5, 0, -0.5>
      scale <6,6,6>
   }
// floor
object{
   polygon 4, <-26,0.-26>, <-26,26>, <26,0,26> <26,0,-26>
   the_floor
   }
// wall's
define wall_seg
object {
   object { box < -5, -1, -21 >, < -4, 8, 21 > stucco }
 + object { box < 4,-1, -21>, < 5, 8, 21> stucco }
 + object { box <-21,-1, -20>, <-4, 8, 21> stucco }
```

CHAPTER THREE

```
continued from previous page
+ object { box < 4,-1, -20>, <21, 8, 21> stucco }
+ object { box <-20.9,-1, -24.9>, <24.9, 0, 29>
   the_walls}
}
// ceiling
object {
   box <-25,8, -20>, <25, 9, 2000>
   the_ceiling { scale <10,10,10> }
define bright_white
texture {
   surface {
      ambient 0.3
      diffuse 0.7
      color <2,2,2>
   }
}
define fixture
object {
   box <-2,7.9,-1>, <2, 8, 1>
   bright_white
light white*dim, < 0,7, 0>
light white*dim, < 0,7, 25>
light white*dim, < 0.7, 50>
light white*dim, < 0.7, 75
light white*dim, < 0,7,100>
light white*dim, < 0.7,125>
light white*dim, < 0.7,150>
light white*dim, < 0,7,200>
light white*dim, < 0.7,250>
light white*dim, < 0.7,300>
light white*dim, < 0,7,350>
light white*dim, < 0,7,400>
light white*dim, < 0.7,450>
light white*dim, < 0.7,500>
light white*dim, < 0,7,550>
light white*dim, < 0.7,600>
light white*dim, < 0,7,650>
light white*dim, < 0.7,700>
light white*dim, < 0.7,750>
light white*dim, < 0,7,800>
light white*dim, < 0,7,850>
light white*dim, < 0.7,900>
light white*dim, < 0.7,950>
fixture {translate <0,0, 0>}
fixture {translate <0,0, 25>}
fixture {translate <0,0, 50>}
```

```
fixture {translate <0,0, 70>}
fixture {translate <0,0,100>}
fixture {translate <0,0,125>}
fixture {translate <0,0,150>}
fixture {translate <0,0,200>}
fixture {translate <0,0,250>}
fixture {translate <0,0,300>}
fixture {translate <0,0,350>}
fixture {translate <0,0,400>}
fixture {translate <0,0,450>}
fixture {translate <0,0,500>}
fixture {translate <0,0,550>}
fixture {translate <0,0,600>}
fixture {translate <0,0,650>}
fixture {translate <0,0,700>}
fixture {translate <0,0,750>}
fixture {translate <0,0,800>}
fixture {translate <0,0,850>}
fixture {translate <0,0,900>}
fixture {translate <0,0,950>}
wall_seg {translate <0,0, 0>}
wall_seg {translate <0,0, 50>}
wall_seg {translate <0,0,100>}
wall_seg {translate <0,0,150>}
wall_seg {translate <0,0,200>}
wall_seg {translate <0,0,250>}
wall_seg {translate <0,0,300>}
wall_seg {translate <0,0,350>}
wall seg {translate <0,0,400>}
wall_seg {translate <0,0,450>}
wall_seg {translate <0,0,500>}
wall seg {translate <0,0,550>}
wall_seg {translate <0,0,600>}
wall_seg {translate <0,0,650>}
wall seq {translate <0,0,700>}
wall_seg {translate <0,0,750>}
wall_seg {translate <0,0,800>}
wall_seg {translate <0,0,850>}
wall_seg {translate <0,0,900>}
wall_seg {translate <0,0,950>}
```

The render time per frame for this model is hours. Reducing it to a single section with mirrors that make it appear to stretch off into never-never land drops the trace time to 25 minutes per frame. The model is shown in Figure 3-10.

In the following Polyray code (FAKE.PI), two views are provided. Changing the value of *overhead_view* from no to yes gives an exterior shot. Conditional processing changes the camera position and determines if the ceiling gets rendered. Use it if you're convinced there's really a tunnel out there:

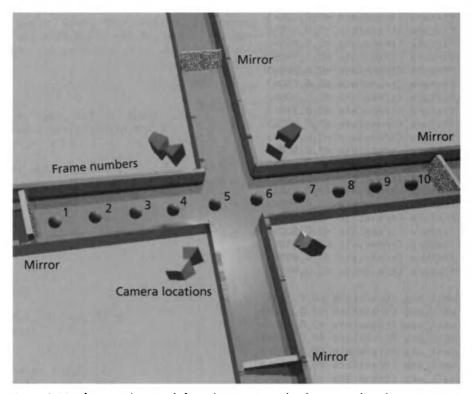


Figure 3-10 Infinite tunnel composed of a single intersecting path with mirrors at the ends

```
// FAKE.PI - Image-Mapped Endless Tunnel with Mirrors
// you need three targa files as image maps (any will do):
11
     the floor - floor.tga
11
     the walls - walls.tga
11
     the ceiling - ceiling.tga
start_frame O
end_frame 9
total_frames 10
outfile "fake"
define no 1
define yes O
define overhead_view no
include "\ply\colors.inc"
if (overhead_view==yes)
```

```
viewpoint {
   from <50,50,-75>
   at <0,0,0>
   up <0,1,0>
   angle 30
   resolution 160,120
   aspect 1.43
   }
}
else {
viewpoint {
   from <0,6,-22.5+frame*5>
   at <0,6,2000>
   up <0,1,0>
   angle 30
   resolution 160,120
   aspect 1.43
}
background <0.4,0.4,0.4>
define disc_image1 image("floor.tga")
define the_floor
   texture {
      special surface {
         color planar_imagemap(disc_image1, P, 1)
         ambient 0.2
         diffuse 0.8
      }
      translate <-0.5, 0, -0.5>
      scale <1,1,1>
   }
define disc image2 image("walls.tga")
define the_walls
   texture {
      special surface {
         color planar_imagemap(disc_image2, P, 1)
         ambient 0.2
         diffuse 0.8
      translate <-0.5, 0, -0.5>
      rotate <0,0,90>
      scale <1,1,1>
   }
define disc_image3 image("ceiling.tga")
define the_ceiling
   texture {
      special surface {
         color planar_imagemap(disc_image2, P, 1)
```

CHAPTER THREE

```
continued from previous page
         ambient 0.2
         diffuse 0.8
      translate <-0.5, 0, -0.5>
      scale <6,6,6>
   }
define reflector
texture {
   surface {
      ambient white, O
      diffuse white, 0
      specular O
      reflection white, 1
      }
   }
// wall's
define wall
   object {
      object { box < -5,-1, -4>, <-4.1, 9, -26> the_walls }
    + object { box < 4.1,-1, -4>, <5, 9,-26> the_walls }
   }
define hall
   object {
       wall {rotate <0, 0,0>}
     + wall {rotate <0, 90,0>}
     + wall {rotate <0,180,0>}
     + wall {rotate <0,270,0>}
}
if(overhead_view==no)
// ceiling
object {
   polygon 4, <-26,8,-26>,<-26,8,26>,<26,8,26>,<26,8,-26>
   the_ceiling
   }
}
// floor
object {
   polygon 4, <-26,0,-26>,<-26,0,26>,<26,0,26>,<26,0,-26>
   the_floor
   }
define bright_white
texture {
   surface {
      ambient 0.3
      diffuse 0.7
      color <2,2,2>
```

```
}
}
define fixture
object {
   box <-3,7.8,-3>, <3, 8, 3>
   bright_white
hall
fixture
fixture {translate < 25,0, 0>}
fixture {translate <-25,0, 0>}
fixture {translate < 0.0, 25>}
fixture {translate < 0,0,-25>}
define dim 0.5
light white*dim,<0,7,0>
light white*dim,<24,7,0>
light white*dim,<-24,7,0>
light white*dim,<0,7,24>
light white*dim,<0,7,-24>
define mirror1
   object {
      polygon 4, <-5,-1,0>,<5,-1,0>,<5,9,0>,<-5,9,0>
      translate <0.0,-25>
      reflector
}
mirror1 {rotate <0, 0,0>}
mirror1 {rotate <0, 90,0>}
mirror1 {rotate <0,180,0>}
mirror1 {rotate <0,270,0>}
```

How It Works

Our sections are 50 units long, running from -25 to 25. Start with the camera at -22.5, then step it 5 units forward each frame. The last frame is at 22.5, again 2.5 units away from the mirror. The next step would place us the equivalent of -22.5 in the next segment, which means the animation will loop perfectly.

This animation is the first one in which a lot of texture maps are used, so a few words about them are in order. Other than obvious stuff like scaling, the only parameter to keep in mind is the angle they make with the item they'll be mapped onto. By default, images map onto the x-z plane. If you

want to map them onto walls, be sure to rotate them about x or z (whichever is appropriate for your model), otherwise you'll map a long thin section of your image across the surface, which may not be what you're after. You can use any image at all for the texture maps, as well as the ones that are included.

Many of the items in this model are defined first, and then copied and used several times. This is not only easy to code, but it also ensures that the model will repeat exactly as it should. The model is simply four corridors rotated about a central space. The rotation is an important part of the modeling. If the walls and floor have conspicuous texture maps, they will not tile properly unless the pattern meets at the mirror points, and a rotation is the simplest way of achieving this. It also helps to have a neutral texture. It makes hiding the mirror points easier.

Comments

You can set the number of times Polyray reflects scenes off mirrors with the *max_level* parameter in the POLYRAY.INI file. It's usually 10 or 20, and it definitely matters in this sort of image. If you want to approach the end of the tunnel and break out into some other animation, try reducing this parameter by one for each series of frames. You'll be able to see the end of the tunnel approaching, and break on through to the other side.



3.6 How do I...

Create a bubbling mud-bog surface?

You'll find the code for this in: PLY\CHAPTER3\BOG

Problem

A few years back, it seemed like almost every high-end math visualization and statistics package used the sombrero function as eye-catchers in their ads to show how good they were at making neat 3-D meshes (Figure 3-11). It's a nice shape, but it basically only does one thing. How would you make an extended surface pocketed with these functions erupt at staggered intervals?

Technique

The answer is, you play with the math and you get lucky. Trigonometry is neat stuff. It deals with the ratios of the length of the sides of triangles, but

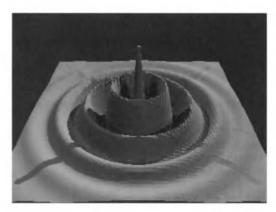


Figure 3-11 The sombrero function

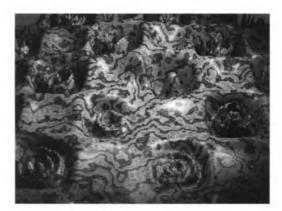


Figure 3-12 Bog surface

the factors involved do wonderful things. Take, for instance, the following equation:

$$sin(2*(r+frame)(cos(x)*sin(z)))*sin(a*4*rad))$$
 where
$$r = (x^2+z^2)^{0.5}$$

$$a = acos(x/r)$$

Know what it does? Neither do I. Haven't the foggiest. It came from an afternoon session of playing with equations and seeing the effects of putting sines, cosines, and powers at random positions in an equation. A high boredom threshold comes in handy at times. But seriously, what it actually does is creates a bubbly surface that's really bad on the eyes (see Figure 3-12).

Steps

The following Polyray data file (BOG.PI) generates the entire animation.

```
// Heightfield BOG Animation
// Polyray input file: Jeff Bowermaster
// define range of frames for animated sequence of images
start_frame 0
end_frame 130
total_frames 131
outfile bog
define pi 3.14159
define rad pi/180
// set up the camera
viewpoint {
   from <20,15,0>
   at <2,0,0>
   up <0,1,0>
   angle 25
   resolution 320,200
   aspect 1.43
// get various surface finishes
include "\ply\colors.inc"
//---- Bug Juice & Green from plyston.inc
define Grnt17
texture {
   noise surface {
      color white
      position_fn 1
      lookup_fn 1
      octaves 6
      turbulence 5
      ambient 0.2
      diffuse 0.6
      specular 0.3
      microfacet Reitz 5
      color_map(
        [0.000, 0.303, <0.000, 0.239, 0.000>, 0.000,
        <0.333, 0.294, 0.000>, 0.000]
        [0.303, 0.588, <0.333, 0.294, 0.000>, 0.000,
        <0.000, 0.239, 0.341>, 0.0001
        [0.588, 0.790, <0.000, 0.239, 0.341>, 0.000,
        <0.000, 0.020, 0.000>, 0.000]
        [0.790, 1.001, <0.000, 0.020, 0.000>, 0.000,
        <0.000, 0.239, 0.000>, 0.000])
   }
```

```
}
// set up background color & lights
background MidnightBlue
spot_light <2.8,2.8,2.8>, <0,10,0>, <0,0,0>, 3, 25, 45
spot_light <2.8,2.8,2.8>, <0,10,9>, <0,0,0>, 3, 25, 45
// this code provides a "fade to flat" loop point
if(frame <110) {
  define fade 1
else {
// fade from frames 110 to 130
  define ra 130 - 110
  define fade (cos(pi * (frame - 110) / ra) + 1) / 2
define time 2*pi*(frame-50)/30
define r (x^2+z^2)^0.5
define a acos(x/r)
define HFn fade*8 * (sin(2 * (r+time) ^ (cos(x) * sin(z))) * sin(a * 4 * rad))
//define detail 50
define detail 200
// define a bog surface, and just take what's in the spotlight by cutting it by
a cylinder
object {
  object {
        smooth_height_fn detail, detail, -15, 15, -15, 15, HFn
        }
 & object {
      cylinder <0, -5, 0>, <0, 5, 0>, 10
  }
```

The key to this entire animation is three lines that specify the heightfield:

```
define r (x^2+z^2)^0.5
define a acos(x/r)
define HFn fade*8 *(sin(2 *(r+time) ^ (cos(x)*sin(z))) * <math>sin(a*4*rad)
```

The animation starts out as a flatline (a featureless plane). This is because the expression for time:

define time 2*pi*(frame-50)/30

is negative up until frame 50, so the expression

```
(r+time) ^ (cos(x) * sin(z))
```

involves exponentiation of a negative number by a non-integer (a major math error under ordinary circumstances). Fortunately, Polyray handles this error by returning 0 for the height. With r equaling the radius of the surface from the origin, as soon as the frame count grows high enough, the surface starts growing from the outside in, eventually converging at the center by frame 50. We use the conditional expression

```
if(frame <110) {
    define fade 1
}
else {
// fade from frames 110 to 130
    define ra 130 - 110
    define fade (cos(pi * (frame - 110) / ra) + 1) / 2
}</pre>
```

to allow the surface generated by the heightfield to be full on up until frame 110, then smoothly fade the heightfield away to a flat plane between frames 110 and 130, resulting in a nice loop back to how we started.



3.7 How do I...

Randomly materialize an object to show its construction?

You'll find the code for this in: PLY\CHAPTER3\FLICKRPOT

Problem

What do you do when you want to see parts of objects that are inside other parts or are hiding behind foreground elements in a scene? You could make the foreground objects transparent and ghost the scene. You could randomly dither the foreground elements (render only some of the pixels that define their surfaces) to allow the background elements to show through. You might even animate the process. In this animation, we'll randomly materialize parts of an object into a rotating scene and use the mind's ability to remember what it saw to show the relationship between the interior and the exterior parts of an object.

Technique

One of the Polyray sample data files is the famous Utah Teapot, a classic shape that's a snap to build with Bezier patches, especially when someone else has done all the work for you. Bezier patches are curved surfaces specified by a series of control points, great for generating objects with complex geometries. The Utah Teapot is composed of 32 patches, which normally appear as a set. For this animation, we'll display random subgroups of them while rotating the collection above a shiny blue table. This resembles whatever it is The Traveler does on Star Trek, the Next Generation, when he's making the Enterprise's warp engines do long division.

Batch File Animations

The typical way to generate animations from ray tracers that don't have internal animation support is to write a batch file that sequentially creates an include file containing the variables that change from frame to frame. This batch program then calls the ray tracer. While we've avoided it up until now, we'll going to use it here, because it makes this particular animation much simpler to write.

Steps

We'd like 32 objects, randomly, to go... please. The following code (RANDOM.BAS) constructs a 4×8 tile grid, picks ten tiles at random, then turns them red. This process repeats until a key is pressed. It demonstrates the random selection process. In the next step, we'll use RANDOM.BAS to generate data for the animation.

```
' RANDOM.BAS
SCREEN 12
WINDOW (-1, -2)-(9, 5)
DIM ox(100), oy(100)
visible = 10
dir = 1
DO WHILE INKEY$ = ""
   'undraw the old ones
   FOR vis = 0 TO visable
      LINE (ox(vis) + .1, oy(vis) + .1) - (ox(vis) + .9, oy(vis) + .9), 1, BF
   NEXT vis
   ' draw the new ones
   FOR vis = 1 TO visible
      x = INT(8 * RND)
      y = INT(4 * RND)
      LINE (x + .1, y + .1)-(x + .9, y + .9), 4, BF
      ox(vis) = x
      oy(vis) = y
   NEXT vis
```

```
continued from previous page
   ' stick around for a sec
   FOR w = 1 TO 10000: NEXT w
LOOP
```

We now have to relate the individually lit squares to the individual parts that make up the teapot. The TEAPOT.INC file is first broken into 32 individual parts, then renamed PARTS.INC. It should take you about 10 minutes to do with any decent editor (although you really don't need to, since PARTS.INC is included on the disc.) This breaks each patch into its own object, which can be called individually.

```
define part00
   object {
      bezier 1, 0.01, 3, 3,
         <1.4,2.4,0>, <1.4,2.4,-0.784>, <0.784, 2.4, -1.4>, <0, 2.4, -1.4>,
         <1.3375, 2.53125, 0>, <1.3375, 2.53125, -0.749>, <0.749, 2.53125, \( \epsilon \)
-1.3375>, <0, 2.53125, -1.3375>,
         <1.4375, 2.53125, 0>, <1.4375, 2.53125, -0.805>, <0.805, 2.53125, \Leftarrow
-1.4375>, <0, 2.53125, -1.4375>,
         <1.5, 2.4, 0>, <1.5, 2.4, -0.84>, <0.84, 2.4, -1.5>, <0, 2.4, -1.5>
 define part01
   object {
      bezier 1, 0.01, 3, 3,
         <0,2.4,-1.4>, <-0.784,2.4,-1.4>, <-1.4,2.4,-0.784>, <-1.4,2.4, 0>,
         <0, 2.53125, -1.3375>, <-0.749, 2.53125, -1.3375>, <-1.3375, 2.53125, \Leftarrow
-0.749>, <-1.3375, 2.53125, 0>,
         <0, 2.53125, -1.4375>, <-0.805, 2.53125, -1.4375>, <-1.4375, 2.53125, \Leftarrow
-0.805>, <-1.4375, 2.53125, 0>,
         <0, 2.4, -1.5>, <-0.84, 2.4, -1.5>, <-1.5, 2.4, -0.84>, <-1.5, 2.4, 0>
. . .
```

A batch file creation program is generated that calls groups of patches in a random order. The script for this file appears in the following listing (SEQUENCE.BAS).

```
' SEQUENCE.BAS
SCREEN 12
WINDOW (-1, -2)-(9, 5)
DIM ox(100), oy(100)
visible = 1
dir = 1
OPEN "seque.bat" FOR OUTPUT AS #1

DO WHILE frame < 186
  visible = visible + 1 * dir
  IF visible = 32 THEN dir = -1
  IF visible = 1 THEN dir = 1

FOR vis = 1 TO visible
```

```
LOCATE 1, 1: PRINT USING "### ## "; frame, visable
      x = INT(8 * RND)
      y = INT(4 * RND)
      LINE (x + .1, y + .1)-(x + .9, y + .9), 4, BF
      ox(vis) = x
      oy(vis) = y
      part$ = "part" + RIGHT$("00" + LTRIM$(STR$(4 * x + y)), 2)
      outfile$ = "tea" + RIGHT$("000" + LTRIM$(STR$(frame)), 3) + ".tga"
      IF vis = 1 THEN
         PRINT #1, "rem frame #"; frame; " >partz.inc"
         PRINT #1, "echo define framer "; frame / 186 * 360; " >>partz.inc"
         PRINT #1, "echo define teapot >> partz.inc"
         PRINT #1, "echo object { >>partz.inc"
         PRINT #1, "echo
                             "; part$; " >>partz.inc"
      ELSE
         PRINT #1, "echo + "; part$; " >>partz.inc"
      END IF
      IF vis = visible THEN
         PRINT #1, "echo shiny_coral >>partz.inc"
         PRINT #1, "echo } >>partz.inc"
         PRINT #1, "\ply\polyray flickpot.pi -o "; outfile$
      END IF
   NEXT vis
   'FOR w = 1 TO 10000: NEXT w
   'DO WHILE INKEY$ = "": LOOP
   FOR vis = 0 TO visible
      LINE (ox(vis) + .1, oy(vis) + .1)-(ox(vis) + .9, oy(vis) + .9), 1, BF
  NEXT vis
   frame = frame + 1
L00P
CLOSE #1
```

SEQUENCE.BAS creates data for 186 frames. This number was selected because it provides a smooth loop point when the number of visible objects is ramped from 1 to 32 then back to 1, three times. An example of the batch code this program generated for frame 9 follows:

```
rem frame # 9 >partz.inc
echo define framer 17.41935 >>partz.inc
echo define teapot >> partz.inc
echo
      object { >>partz.inc
         part08 >>partz.inc
echo
echo
       + part07 >>partz.inc
echo
       + part29 >>partz.inc
      + part16 >>partz.inc
echo
echo
      + part20 >>partz.inc
echo
       + part31 >>partz.inc
echo
      + part16 >>partz.inc
echo
      + partO1 >>partz.inc
echo
      + part27 >>partz.inc
```

CHAPTER THREE

The echo ... >> (redirection to a file) DOS operator writes the file PARTZ.INC, which looks like this:

PARTZ.INC is called inside the following flickering teapot animation routine using this line

```
// get the specific parts
include "partz.inc"
```

Figure 3-13 shows us what frame 9 looks like. The entire listing for the flickering teapot Polyray data file follows.

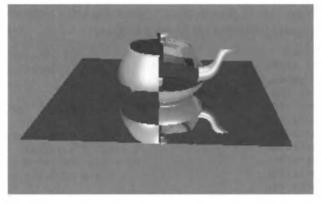


Figure 3-13 Frame 10 of our flickering teapot animation

```
// Flickering Teapot
// Polyray input file: Jeff Bowermaster
// set up the camera
// get various surface finishes
include "\ply\colors.inc"
// set up background color & lights
background midnightblue
light <10, 10, -20>
light <-10, 10, -20>
light <0, 20, 0>
// get the definition of a teapot
include "parts.inc"
// get the specific parts
include "partz.inc"
define vw rotate(<6,8,-16>,<0,framer,0>)
viewpoint {
   from vw
       <0,-0.5, 0>
  at
      <0,1,0>
  up
   angle 30
  resolution 320,200
  aspect 1.43
// make a tabletop
define table
object {
  polygon 4,<-5,0.01,-5>,<5,0.01,-5>,<5,0.01,5>,<-5,0.01,5>
  reflective_blue
}
object {
  box <-5,-1,-5>,<5,0,5>
   texture {surface { color navyblue ambient 0.2 diffuse 0.6}}
}
teapot
table
```

We've already covered the fundamentals of how this animation works in the Steps section, so we'll review the essentials. The global parts of the teapot are included, using the code:

// get the definition of a teapot
include "parts.inc"

This is handled as definitions, not as actual items that would render. The way to render them is to call them by name. A list of the parts we'd like to see in each image resides in the file called PARTZ.INC. It's a random list of names of these parts, created by the QuickBasic program, which does the job of actually throwing them together in the image. Calling them by name brings them to light. A rotating viewpoint is created, and the animation runs.



3.8 How do I...

Show cell division and crystal growth?

You'll find the code for this in: PLY\CHAPTER3\CELL

Problem

The next two animations are going to involve changing the positions of a camera to coincide with the growth of an object. When timed correctly, it appears that the vantage point is fixed, but the object is becoming more detailed. The goal is to make it look like one sphere smoothly subdivides into lots and lots of smaller spheres, while it's actually a growing cluster of spheres seen from a receding vantage point.

Technique

We start off with a complete 3-D array of spheres, say oh, 1,200 of them, focus in on one, and then grow a decision sphere out from that point. The decision sphere determines which spheres are visible; if they're within a certain distance from the origin they appear (see Figure 3-14). The camera zooms back to match the rate of growth of the decision sphere, so that more and more spheres come into view, but because the vantage point recedes, the size of the cluster remains constant. If taken far enough, there would eventually be so many smaller spheres that the figure would coalesce into a single sphere again.

Unfortunately, starting it off is a bit of a problem. There's a sudden 100% increase in size going from one sphere to two, which doesn't quite match the smooth pull back of the camera. It's like starting a car in gear. To mask this problem, we create one big sphere that surrounds the first few spheres as they cluster and melt it back into the cluster once it's growing smoothly on its own. This is known in the graphics trade as "spackle." The spackle sphere

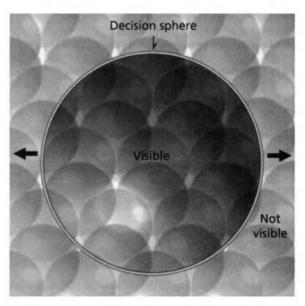


Figure 3-14 Hex grid showing the expanding decision sphere

is controlled by the variable *popper*, reaches a maximum size at frame 30, then recedes.

Steps

We'll start with a hexagonal array of spheres, as illustrated in Figure 3-14. The centers of the spheres can be thought of as ping pong balls imbedded in chicken wire. The structure would then be made of alternating layers of chicken wire in which centers of one layer line up with the intersections of the next layer. To generate it requires a lot of square roots of 3. Modulo 2 math handles the alternating aspect of the layers. It's hideous but harmless. The following code (HEXCELL1.PI) generates appropriate header information for the animation and the hex grid with decision code that is added to each cell:

```
OPEN "Hexcell1.PI" FOR OUTPUT AS #1

PRINT #1, "// Growing Hexagonal Cluster That Tumbles"
PRINT #1,
PRINT #1, "start_frame 1"
PRINT #1, "end_frame 180"
PRINT #1, "total_frames 180"
PRINT #1,
PRINT #1,
PRINT #1, "define index frame*360/total_frames"
```

```
continued from previous page
PRINT #1, "define size 12*frame/total_frames"
PRINT #1, "define pi 3.14159"
PRINT #1, "define rad pi/180"
PRINT #1, "define tumble <index,45*sin(2*index*rad),90*cos(index*rad)>"
PRINT #1,
PRINT #1, "outfile "; CHR$(34); "hex"; CHR$(34)
PRINT #1,
PRINT #1, "include "; CHR$(34); "\pty\colors.inc"; CHR$(34)
PRINT #1,
PRINT #1, "viewpoint {"
PRINT #1, " from rotate(<15,35,-20>, tumble)/12*(size+1)"
PRINT #1, " at <0,0,0>"
PRINT #1, " up <0,1,0>"
PRINT #1, " angle 40"
PRINT #1, " resolution 320,200"
PRINT #1, " aspect 1.333"
PRINT #1, " }"
PRINT #1,
PRINT #1, "background MidnightBlue"
PRINT #1,
PRINT #1, "spot_light <1,0,0>, < 100,0,0>,<0,0,0>,3,5,20"
PRINT #1, "spot_light <0,1,0>, <-100,0,0>,<0,0,0>,3,5,20"
PRINT #1, "spot_light <0,0,1>, < 0, 100,0>,<0,0,0>,3,5,20"
PRINT #1, "spot_light <1,1,0>, < 0,-100,0>,<0,0,0>,3,5,20"
PRINT #1, "spot_light <0,1,1>, < 0,0, 100>,<0,0,0>,3,5,20"
PRINT #1, "spot_light <1,0,1>, < 0,0,-100>,<0,0,0>,3,5,20"
PRINT #1,
PRINT #1, "define pop1 3*frame/30"
PRINT #1, "define pop2 3*sin(3*frame*rad)"
PRINT #1,
PRINT #1, "if (frame < 30) {"
PRINT #1, " object { sphere <0,0,0>,pop1 matte_white }"
PRINT #1, "}"
PRINT #1,
PRINT #1, "if (frame < 60) {"
PRINT #1, "
            object { sphere <0,0,0>,pop2 matte_white }"
PRINT #1, "}"
PRINT #1,
cel = 8
n = 0
FOR z = -cel TO cel
  z1 = z * 1.632993 + 0
                           ' sqrt(8/3)z
   alt1 = 1
  IF z \mod 2 = 0 THEN alt1 = 0
   FOR y = -cel TO cel
      y1 = y * 1.732051 - 1.154701 * alt1 ' sqrt(3)y + 2*sqrt(3)/3
     alt2 = 1
     IF y MOD 2 = 0 THEN alt 2 = 0
```

```
FOR x = -cel TO cel
         x1 = x * 2 + 1 * alt2  2x + 1 or 2x + 0
         IF (x1 ^2 + y1 ^2 + z1 ^2) ^3.5 < 12 THEN
            vect$ = "c" + RIGHT$("0000" + LTRIM$(STR$(n)), 4)
            PRINT #1, USING "define \ \ < ###.#####,###.#########"; ←
vect\$, x1, y1, z1
            PRINT #1, USING "if ( |\ \| < size) {"; vect$
            PRINT #1, USING " object { sphere \ \, 1 matte_white } "; ←
vect$
            PRINT #1, "}"
            n = n + 1
         END IF
      NEXT x
   NEXT y
NEXT z
CLOSE #1
           The output of HEXCELL1.PI is as follows:
// Hexagonal Cluster That Tumbles
start_frame 1
end frame 180
total_frames 180
define index frame*360/total_frames
define size 12*frame/total_frames
define pi 3.14159
define rad pi/180
define tumble <index,45*sin(2*index*rad),90*cos(index*rad)>
outfile "hex"
include "\ply\colors.inc"
viewpoint {
   from rotate(<15,35,-20>, tumble)/12*size
   at <0.0.0
   up <0,1,0>
   angle 40
   resolution 320,200
   aspect 1.333
background MidnightBlue
spot_light <1,0,0>, < 100,0,0>,<0,0,0>,3,5,20
spot_light <0,1,0>, <-100,0,0>,<0,0,0>,3,5,20
spot_light <0,0,1>, < 0, 100,0>,<0,0,0>,3,5,20
spot_light <1,1,0>, < 0,-100,0>,<0,0,0>,3,5,20
spot_light <0,1,1>, < 0,0, 100>,<0,0,0>,3,5,20
                                                                continued on next page
```

```
continued from previous page
spot_light <1,0,1>, < 0,0,-100>,<0,0,0>,3,5,20
define pop1 3*frame/30
define pop2 3*sin(3*frame*rad)
if (frame < 30) {
   object { sphere <0,0,0>,pop1 matte_white }
if (frame < 60) {
   object { sphere <0,0,0>,pop2 matte_white }
}
define c0000 < -1.00000, -2.88675,-11.43095>
if ( |c0000| < size) {
   object { sphere c0000, 1 matte white }
}
define c0001 < 1.00000, -2.88675,-11.43095>
if ( |c0001| < size) {
   object { sphere c0001, 1 matte_white }
define c0002 < -2.00000, -1.15470,-11.43095>
if ( |c0002| < size) {
   object { sphere c0002, 1 matte_white }
}
define c0003 < 0.00000, -1.15470,-11.43095>
if (|c0003| < size) {
   object { sphere c0003, 1 matte_white }
}
define c0004 < 2.00000, -1.15470,-11.43095>
if (|c0004| < size) {
   object { sphere c0004, 1 matte_white }
define c0005 < -3.00000, 0.57735,-11.43095>
if (|c0005| < size) {
  object { sphere c0005, 1 matte_white }
}
define c0006 < -1.00000, 0.57735,-11.43095>
if ( |c0006| < size) {
   object { sphere c0006, 1 matte_white }
}
define c0007 < 1.00000, 0.57735,-11.43095>
if ( |c0007| < size) {
   object { sphere c0007, 1 matte_white }
define c0008 < 3.00000, 0.57735,-11.43095>
if ( |c0008| < size) {
  object { sphere c0008, 1 matte_white }
}
. . .
```

An example of what we're generating is shown in Figure 3-15.

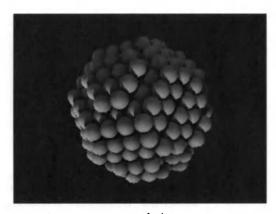


Figure 3-15 A growing mass of spheres

The growth is handled using Polyray's vector length function *lvexperl*. The variable *size* grows outward each frame

```
define size 12*frame/total_frames
```

and sets the size of the decision sphere. If the length of the vector defining the position of the sphere is less than *size*, the object sphere will pop into view in our image.

The code created by this QuickBasic program is rather lengthy, as there are 1,250 spheres in the array. The listing was abbreviated.

Two Decision Spheres

This animation renders quickly, but it's still not quite as smooth as we'd like. A smoother way of growing the sphere without "spackle" requires using two decision spheres, one twice the size of the other. The second sphere brings in elements before they would normally appear in the scene, but scaled down so that they don't overtake the primary sphere growth. They fill in the cracks. As the decision sphere expands, we enlarge and position these spheres back to their original size. They then take the positions they would have if we had used a single decision sphere. This is all handled by a second *if* clause shown in this excerpt:

```
define c0000 < -1.00000, -2.88675,-11.43095>
if ( |c0000| < size) {
  object { sphere c0000, 1 matte_white }
}
// 2 times the previous decision sphere
if ( |c0000| < size*2) {
  object { sphere c0000*size/|c0000|, 1 matte_white }
}</pre>
```

If the next sphere is exactly two times further away from being picked for view, it's scaled into the current viewable area by multiplying its position by 0.5, since *size* is then half the value of |c0000|. As *size* grows, the position of the sphere is scaled less and less. In fact, it tracks the surface growth exactly. Once it's caught by the original decision sphere, it's back where it belongs. Basically, this results in a phased introduction of offset spheres that makes for smoother growth. The code for this is found in HEXCELL2.BAS.

Crystal Growth

A variation of the cell division code produces a nice hex crystal growth animation. Its decision statement really only kicks in at the beginning of the animation, smoothing the formation point. Once the decision sphere size grows, the second spheres get lost in the grid

```
define c0000 < -1.00000, -2.88675,-11.43095>
if ( |c0000| < size) {
   object { sphere c0000, 1 matte_white }
}
if ( |c0000| < size*2) {
   object { sphere c0000*2*size/12, 1 matte_white }
}</pre>
```

Comments

The cluster is defined as individual spheres, so tumbling the cluster would require individual rotate statements for each sphere. Although this could be done with QuickBasic, a simpler approach is to tumble the camera. The *from* expression has three parts. It combines the static view from <15,35,-20>, which is rotated by a dynamic vector *tumble*, and then scaled so that it zooms away as the *size* value grows.



3.9 How do I...

Eat away at a complex object to show its interior form?

You'll find the code for this in: PLY\CHAPTER3\POP

Problem

If you wanted to define a sponge, you might consider doing so by throwing together a bunch of spheres and then fusing them together into an amorphous mass. There'd have to be a "field" set up around each sphere so

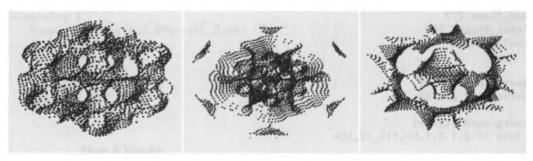


Figure 3-16 Output-interior details of blobs defined by an array of spheres

that you could add the fields together to define the structure as a whole. As you'll recall from Section 2.4, Blobs, one of Polyray's graphics primitives, work much like this. The fields surrounding the spheres each contribute to the value assigned to any point in space, and a blob's surface traces out regions where this value is equal to some constant. A result of this definition is that blobs can have some interesting interior structures, hidden below their lumpy exterior forms. Tunnels, caves and little arches exist on their insides. This animation cuts through the exterior layers to reveal these interior structures (see Figure 3-16).

Technique

The blob is defined by a central sphere, eight spheres arranged like the corners of a cube surrounding this sphere, and 12 midpoints of lines between these corners. Blobs are covered in more detail in Chapter 7, *Blobs*. A threshold value allows adustment of the overlap amount of the spheres that define the form. Adjusting the threshold to the correct value generates the appropriate interior details.

As in the previous example, we synchronize the motion of the camera with the clipping sphere, so that it looks as if the shape is dynamically evolving, while in fact it's the viewpoint that's moving.

Steps

Run the following Polyray data file (POP.PI).

```
// POP.PI
// Cut Away Blob Shape
include "\ply\colors.inc"
start_frame 0
```

CHAPTER THREE

```
continued from previous page
end_frame 179
total_frames 180
outfile "bls"
define pi 3.14159
define rad pi/180
background SkyBlue
light <1.0,1.0,1.0>,<10,12,10>
define weird_shape
object {
   blob 6.6:
     7, 3.0,< 0, 0, 0 >,
     3, 1.0,< 1, 1, 1 >,
     3, 1.0, < -1, 1, 1 >,
     3, 1.0,< 1, -1, 1 >,
     3, 1.0, < -1, -1, 1 >
     3, 1.0,< 1, 1, -1 >,
     3, 1.0, < -1, 1, -1 >,
     3, 1.0,< 1, -1, -1 >,
     3, 1.0, < -1, -1, -1 >,
     3, 1.0,< 0, 1, 1 >,
     3, 1.0, < 0, 1, -1 >
     3, 1.0,< 0, -1, 1 >,
     3, 1.0, < 0, -1, -1 >
     3, 1.0, < 1, 0, 1 >
     3, 1.0, < 1, 0, -1 >,
     3, 1.0, < -1, 0, 1 >
     3, 1.0, < -1, 0, -1 >
     3, 1.0,< 1, 1, 0 >,
     3, 1.0, < 1, -1, 0 >
     3, 1.0,< -1, 1, 0 >,
     3, 1.0, < -1, -1, 0 >
   root_solver Sturm
   u_steps 20
   v_steps 20
   reflective_coral
define ang 360*frame/total_frames
// range needs to be 0.52 to 1.66
define breath 0.57*cos(rad*ang)+1.09
object {
  weird_shape
& object { sphere <0,0,0>, breath } // clip it by a breathing sphere
   rotate <0, ang,0>
   }
```

```
viewpoint {
   from <1.3*breath,1.8*breath,0.0>
   at <0.0,0.0,0.0>
   up <0,1,0>
   angle 50
   resolution 320,200
   aspect 1.433
}
```

We generate a blob and call it weird_shape. We define a variable ang that goes from 0 to 360 regardless of the frame count, and a breath function that goes from 0.52 to 1.66. The blob either completely disappears or is fully visible outside that range. As the blob shape rotates, we clip it by the breathing shape and synchronize the motion of the camera with the size of the clipping sphere so that the shape appears to change rather than simply being cut away.

Moose

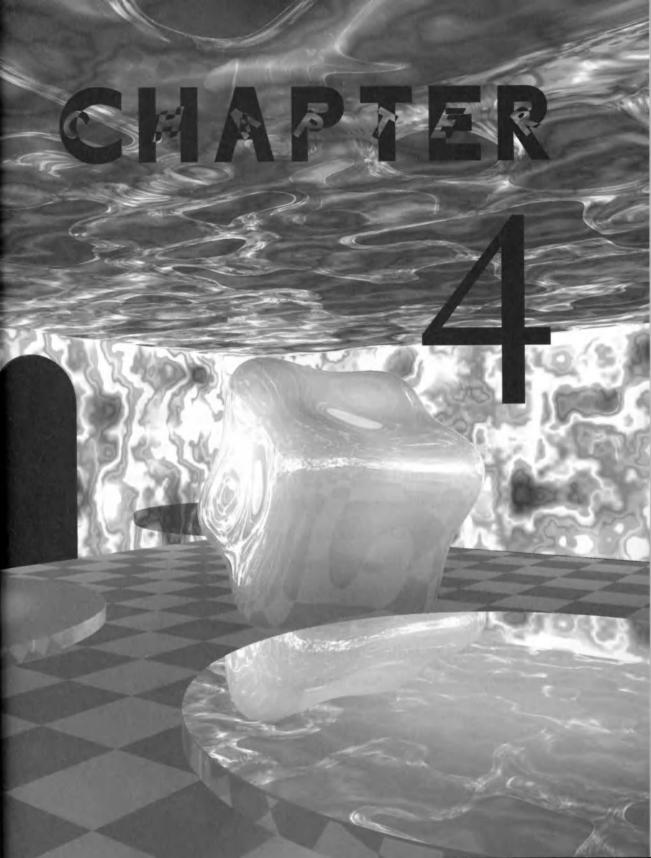
As luck would have it, one of the cut-away views of this blob just happened to form something resembling the head of a moose (see Figure 3-17). Red glowing eyes were added to distinguish it from Bullwinkle. A rotating animation, showing just how much it doesn't resemble a moose from other angles, is provided as MOOSE.PI. The script for the animation is as follows:



Figure 3-17 Not Bullwinkle

```
// MOOSE.PI
// Moose Shaped Blob Reflection Deal
include "\ply\colors.inc"
start_frame 0
end_frame 89
total_frames 90
define index 360/total_frames
outfile "moose"
viewpoint {
   from <0.8,0.8,0.0>
   at <0.0,0.0,0.0>
   up <0,1,0>
   angle 90
   resolution 320,200
   aspect 1.433
   }
background SkyBlue
light <0.7,0.7,0.7>,<10,12,1>
define ang frame*index
define unit_sphere object { sphere <0, 0, 0>, 1.0 }
define eye
   object {
      object {
         sphere <0.0,0.0,0.0>,0.10
         texture {
            surface {
               ambient red, 0.2
               diffuse red, 0.5
               specular white, 0.7
            }
         }
      }
   + object {
         sphere <0.0,0.0,0.0>,0.05
         texture {
            surface {
               ambient Yellow, 0.5
               diffuse Coral, 0.8
               specular white, 0.9
            }
         }
      translate <0.05,0.05,0.0 >
   }
define weird_shape
```

```
object {
   blob 6.6:
      7, 3.0,< 0, 0, 0 >,
      3, 1.0,< 1, 1, 1 >,
      3, 1.0, < -1, 1,
                       1 >,
      3, 1.0, < 1, -1,
                      1 >,
      3, 1.0, < -1, -1, 1 >
      3, 1.0,< 1, 1, -1 >,
      3, 1.0, < -1, 1, -1 >
      3, 1.0, < 1, -1, -1 >
      3, 1.0, < -1, -1, -1 >
      3, 1.0,< 0, 1, 1 >,
      3, 1.0, < 0, 1, -1 >
      3, 1.0,< 0, -1, 1 >,
      3, 1.0, < 0, -1, -1 >
      3, 1.0, < 1, 0, 1 >
      3, 1.0, < 1, 0, -1 >
      3, 1.0, < -1, 0,
                      1 >,
      3, 1.0, < -1, 0, -1 >
      3, 1.0,< 1, 1, 0 >,
      3, 1.0, < 1, -1, 0 >
      3, 1.0,< -1, 1, 0 >,
      3, 1.0, < -1, -1, 0 >
   root_solver Sturm
   u_steps 20
   v_steps 20
   reflective_coral
object {
   weird_shape & unit_sphere
   rotate <45.0,ang,0.0>
   }
eye {
    rotate <70,70,30>
    translate <0.25,0.25,0.25>
    rotate <0.0, ang, 0.0>
eye {
    rotate <10,-10,0>
    translate <0.25,0.25,-0.25>
    rotate <0.0,ang,0.0>
    }
```



4

MR. WIGGLY

great many animations derive their special appeal from the periodic application of three-dimensional shape changes. A stone cast into a pond ripples the surface. Jellyfish swim about with a fluid grace. Manta rays ripple their edges. All these motions involve defining objects in sections and generating phased, coordinated motions of their component parts. All these animations use a series of variables to control the phase and amplitude of the motions. It's a simple matter to change these variables to get whatever motion you desire. A fringe benefit of wiggling animations is that they always loop nicely.



邎 4.1 How do I...

Animate ripples on a pond?

You'll find the code for this in: PLY\CHAPTER4\IMPACT

Problem

The spreading ring of waves that occurs when a stone is dropped into a still pool of water requires specifying a time sequence for the spreading wave front in terms of its surface height and diameter. This is best accomplished using a functional heightfield generated by rotating an impulse wave about the origin. A functional heightfield is a surface whose height (y) can at any point be derived from an equation involving the location on that surface (the position variables x and z).

Technique

Let's closely examine all the processes involved. A ball approaches a surface and, on impact, an indentation forms as the ball passes through it. Fluid rushes in to fill this void and ends up overshooting the initial level. A ring centered on the impact spreads out, and the height of this ring diminishes as it spreads.

The expanding wave front can be modelled as a radially symmetrical Gaussian function. Any simple linear function can be rotated about the origin by substituting its linearly dependent variable (usually x) by the equation for a circle $(r = (x^2 + z^2)^{0.5})$. A functional heightfield specifies the height of a surface using a simple equation containing this radius r

```
define r (x^2+2^2)^0.5
define HFn height / EXP(((r - wave_radius) / width) ^ 2)
```

that defines the expanding ring without us having to mess with the details.

An interesting twist in this animation is that we need to start it with a ring already expanded, shrink it around the ball as it passes through the surface, then let it expand again. The whole timing sequence is handled with IF-ELSE statements based on the frame count. The waveforms and equations needed to control the motions were hand crafted and visualized in QuickBasic, with the goal of smoothly combining linear, sinusoidal, and exponential functions at key junctions. Figure 4-1 shows one frame from our animation, showing the expansion ring, and Figure 4-2 shows the splash radius and the surface height plotted as a function of time.

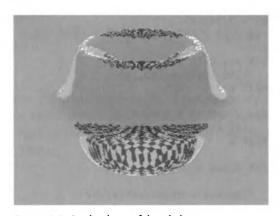


Figure 4-1 Overhead view of the splash ring

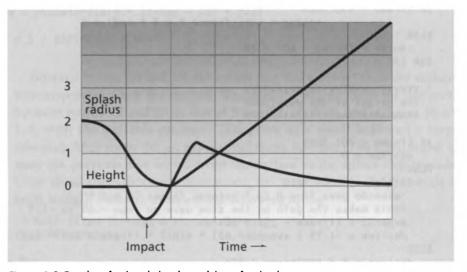


Figure 4-2 Time lines for the splash radius and the surface height

Steps

Interactive QuickBasic visualization code helps generate the motion sequence for the height of the surface and the diameter of the ring. Select a group of functions that might work (have about the right shapes) and plug in coefficients until they connect at their endpoints and look about right. The following program (SPLASH.BAS) creates the curves seen in Figure 4-1:

```
'SPLASH.BAS

SCREEN 12

WINDOW (-1, -1.5)-(141, 2)

LINE (0, -1)-(140, 5), B
```

```
continued from previous page
pi = 3.14159
rad = pi / 180
FOR y = 0 TO 5
  LINE (0, y) - (140, y)
NEXT y
FOR x = 0 TO 140 STEP 20
 LINE (x, -1)-(x, 5)
NEXT x
FOR frame = 0 TO 140
   ' the spreading wavefront
   'AAAAAAAAAAAAAAAAAAAAAA
  IF (frame < 40) THEN
                  center = COS((frame) * 4.8 * rad) + 1
  ELSE
     center = (frame - 40) / 20
  END IF
   ' the height of the wavefront
   IF (frame < 20) THEN
     decline = 0
   ELSE
     IF (frame < 50) THEN
      'expando goes from 0 to 1 between frames 20 and 50
      'this makes the gain on the sine wave go from -.65 to -1.3
      expando = ((frame - 20) / 30)
      decline = -(.75 + expando*.65) * SIN(9 * (frame - 20) * rad)
  ELSE
     decline = 5 / EXP(frame / 40)
   END IF
END IF
CIRCLE (frame, center), .2, 2
CIRCLE (frame, decline), .2, 3
```

The splash radius is controlled by the *center* variable. We start it out at two, collapse it to zero at frame 40, then grow it back linearly with a slope of 1/20 units per frame after frame 40:

NEXT frame

The factors 4.8 and *rad* adjust the wavelength to fit in the space we're using, and assist in lining up the ends. Note that it doesn't matter that the splash radius is nonzero at the start of the animation. The surface height is zero until frame 20, so it doesn't show:

```
IF (frame < 20) THEN
   decline = 0
ELSE
   IF (frame < 50) THEN
   'phasein goes from 0 to 1 between frames 20 and 50
   phasein = -(0.75+(frame - 20) / 30)*0.65)
   decline = phasein*SIN(9 * (frame - 20) * rad)
ELSE
   decline = 5 / EXP(frame / 40)
END IF</pre>
```

Between frames 20 and 50, the splash ring sinks down below the surface, which opens up a hole for the ball. We adjust the magnitude of the sine wave between frames 20 and 50 so that at frame 20, it's 0.65, and at frame 50 it's 1.4, with the variable *phasein*. This gives us a small hole and a large rebound. After frame 50, an exponential decay function (1/exp(x)), scaled to meet the previous sine wave, drops the surface as the splash ring spreads. Once these coefficients are determined, the translation to Polyray code is fairly straight forward, as seen in the following listing:

```
// Impact: A Ball Slips Through a Viscous Mirror
            Jeff Bowermaster
start_frame O
end_frame 134
total_frames 135
outfile imp
// set up the camera
viewpoint {
   from <5,3,5>
   at <0,-0.12,0>
   up <0,1,0>
   angle 25
   resolution 320,240
   aspect 1.33
   }
// get various surface finishes
```

CHAPTER FOUR

```
continued from previous page
include "\PLY\COLORS.INC"
define mirror2
texture {
   surface {
      ambient gold, 0.1
      diffuse white, 0.2
      specular O
      reflection white, 1
      }
   }
// set up background color & lights
background SkyBlue
// a coral colored light
light <1,0.5,0>,<0,10,0>
define pi 3.14159
define rad pi/180
// make the wave front collapse and then expand
if (frame <40)
   define wave_radius cos((frame)*4.8*rad) + 1
else
   define wave_radius (frame-40)/20
define phase_in -(0.75+0.65*(frame-20)/30)
//start with a flat surface, indent and rebound, then decay
if (frame <20)
   define height O
else if (frame <50)
   define height phase_in*sin(9 * (frame - 20) * rad)
else
   define height 5/exp(frame/40)
// heightfield is a radially symmetrical expanding gaussian
define width 0.3
define r (x^2+z^2)^0.5
define HFn height / EXP(((r - wave_radius) / width) ^ 2)
define detail 100
// define the wave surface
object {
     smooth_height_fn detail, detail, -10, 10, -10, 10, HFn
     mirror2
     }
```

```
// drop an expanding ball on it

define ball (23-frame)/7.5
define grow 0.25+0.25*frame/20

object {
    sphere <0,ball,0>,grow
    shiny_coral
    }

// give the surface something hidden to reflect

object {
    disc <0, 20, 0>, <0, 1, 0>, 20
    texture {
        checker matte_white, matte_black
        translate <0, -0.1, 0>
        scale <2, 1, 2>
    }
}
```

The QuickBasic code and the Polyray code are identical in the way they achieve the various stages of this animation. Conditional processing sets the times when key actions occur, like the change in the diameter of the splash ring and its rise and fall.

The surface is defined using the *height* and *wave_radius* variables generated by all this conditional code, defined as the variable *HFn*, which is included at the end of a *smooth_height_fn* definition. The *detail* variable sets the level of smoothness, or the number of subdivisions the heightfield uses to approximate the surface. Polyray calculates heights for a square running from -10 to 10 in both x and z:

```
define width 0.3
define r (x^2+z^2)^0.5

define HFn height / EXP(((r - wave_radius) / width) ^ 2)
define detail 100
// define the wave surface
object {
    smooth_height_fn detail, detail, -10, 10, -10, 10, HFn
    mirror2
}
```

All this motion is coordinated with dropping a ball on the surface. It touches the surface at frame 20, but its center doesn't make it through until frame 23:

```
define ball (23-frame)/7.5
define grow 0.25+0.25*frame/20
```

The ball is visible as a reflection on the surface before it comes into actual view, and to exaggerate the fall, we expand it as it drops with the *grow* variable that sets its radius. This effect was originally added to reduce its size and keep it hidden at frame 0, since without this scaling, it would suddenly pop into view at the loop point. Later on, the camera was moved and this scaling was no longer required. However the expanding effect was interesting, and so it was kept.

Comments

There are a few interesting twists here. The surface is a mirror that when undisturbed fails to reflect a checkerboard that's been placed up above the camera and out of view. It's only seen in the surface ripples. The lighting creates a coral checkerboard, but has no impact on the color of the mirror.

If the fluid overshoots the indentation once, it should be able to do it several times, producing a damped oscillation. We might attempt to increase the complexity of the heightfield by using our Gaussian function to shape a packet of sine waves. It was simpler to assume that the fluid was very viscous and that a single pulse was all we would get, but don't let that stop you from trying this variation.



🎉 4.2 How do l...

Make a diamond swim like a jellyfish?

You'll find the code for this in: PLY\CHAPTER4\JELLYDIM

Problem

Solid objects defined as triangle grids make ideal candidates for 3-D morphs. Coordinated deformations require establishing a framework that collects the points defining the triangles together in groups. These groups are then translated relative to one another using some procedure. The least flexible object in nature is a diamond. It would be interesting to see it swim like a jellyfish.

Technique

If you've ever watched a jellyfish swim, you may have noticed it's not exactly a continuous sine wave motion like a spring. It's more of a heave and a

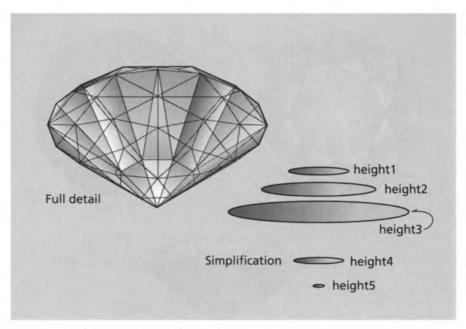


Figure 4-3 The full view and the simplification of a diamond

pause, a heave and a pause. You may have also noticed that different parts of the jellyfish move at different times, so that the relative motions of the parts are *phased*.

We approach this animation in two stages. First, we'll create a real-time simulation of a rough model of our diamond. Then we'll write a diamond generator inside Polyray, exploiting a faceted diamond's eight-way symmetry, and layer the motion on top of this geometry.

Steps

The basic simulation code needs to reduce the complexity of the diamond to a simpler form for a faster running wire frame simulation. A fully detailed diamond wire frame and the simplification we'll use are shown in Figure 4-3.

The simplification breaks the diamond up into five different levels, represented by circles connecting vertices of different levels. A better view of the diamond showing these five different heights is shown in Figure 4-4.

The following code animates the simplified version of the model and allows us to adjust the phase and amplitude of the various components until we're happy with the results:

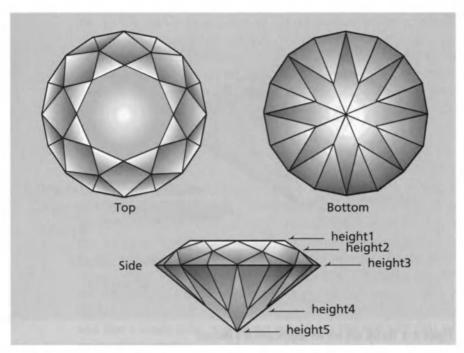


Figure 4-4 The diamond. Coplanar triangles in the full detail view have been collapsed into trapezoids

```
' JellyDim.BAS
' Jellyfish Diamond Motion File
TYPE vector
  x AS SINGLE
  y AS SINGLE
  z AS SINGLE
END TYPE
pi = 3.1415926536#
radians = 180 / pi
DIM vertex(42) AS vector, triangle(3) AS vector
DIM red(16), green(16), blue(16)
' set the screen up with pretty rainbow colors
SCREEN 12
zoom = .015
xoff = 320
yoff = 240
WINDOW ((0 - xoff) * zoom, (0 - yoff) * zoom)-((639 - xoff) * zoom, (479 - \Leftarrow
yoff) * zoom)
FOR y = 1 TO 4
```

```
FOR x = 1 TO 4
               colornum = x + ((y - 1) * 4) - 1
               READ red(colornum), green(colornum), blue(colornum)
               KOLOR = 65536 * blue(colornum) + 256 * green(colornum) + ←
red(colornum)
               PALETTE colornum, KOLOR
               COLOR colornum
       NEXT x
NEXT y
'rainbow palette
DATA 0, 0, 0
DATA 32, 0, 0
DATA 42, 0, 0
DATA 58, 16, 0
DATA 63, 32, 0
DATA 58, 56, 0
DATA 16, 42, 0
DATA 0, 30, 36
DATA 0, 20, 40
DATA 0, 10, 48
DATA 0, 0, 63
DATA 20, 0, 53
DATA 23, 0, 29
DATA 19, 7, 17
DATA 50, 40, 45
DATA 63, 63, 63
pi = 3.1415926535#
rad = pi / 180
radius1 = 1
radius2 = 1.4
radius3 = 1.8
radius4 = .7
radius5 = .01
ht1 = 1
ht2 = .8
ht3 = .5
ht4 = -.5
ht5 = -1
masteramp = 1.3
ampl1 = .03 * masteramp
ampl2 = .02 * masteramp
ampl3 = .01 * masteramp
ampl4 = .04 * masteramp
ampl5 = .05 * masteramp
phase = 80 ' hump phase
```

CHAPTER FOUR

```
continued from previous page
phase1 = 0
phase2 = 25
phase3 = 50
phase4 = 75
phase5 = 87
aspect = .1 ' control the size of the ellipses in the simplified model
bump = 1
wave = 4
DO WHILE INKEY$=""
   ang = ang + 10
  height1 = ht1
          + ampl1 * (wave * COS(rad * (ang - phase1))
          + EXP(bump * (1 + COS((ang - phase - phase1) * rad))))
  height2 = ht2
          + ampl2 * (wave * COS(rad * (ang - phase2))
          + EXP(bump * (1 + COS((ang - phase - phase2) * rad))))
          + ampl3 * (wave * COS(rad * (ang - phase3))
          + EXP(bump * (1 + COS((ang - phase - phase3) * rad))))
  height4 = ht4
          + ampl4 * (wave * COS(rad * (ang - phase4))
          + EXP(bump * (1 + COS((ang - phase - phase4) * rad))))
  height5 = ht5
          + ampl5 * (wave * COS(rad * (ang - phase5))
          + EXP(bump * (1 + COS((ang - phase - phase5) * rad))))
  newradius1 = radius1 * height1
  newradius2 = radius2 * height2 * 1.3
  newradius3 = radius3 * height3 * 2
  newradius4 = radius4 * (1.5 + height4)
  newradius5 = radius5
  CIRCLE (0, h1old), r1old, 0, , , aspect
  CIRCLE (0, h2old), r2old, 0, , , aspect
  CIRCLE (0, h3old), r3old, 0, , , aspect
  CIRCLE (0, h4old), r4old, 0, , , aspect
  CIRCLE (0, h5old), r5old, 0, , , aspect
  CIRCLE (0, height1), newradius1, 2, , , aspect
  CIRCLE (0, height2), newradius2, 3, , , aspect
  CIRCLE (0, height3), newradius3, 4, , , aspect
  CIRCLE (0, height4), newradius4, 5, , , aspect
  CIRCLE (0, height5), newradius5, 6, , , aspect
  h1old = height1
  h2old = height2
  h3old = height3
  h4old = height4
  h5old = height5
```

```
r1old = newradius1
r2old = newradius2
r3old = newradius3
r4old = newradius4
r5old = newradius5
```

In the simplified diamond model, the five rings oscillate up and down under the control of five *height* equations. The top level *height1* is defined as

This equation is broken into three terms, which we'll cover next. Note that QuickBasic requires that all equations fall on a single line, but to fit this equation onto the page it had to be split up, and it makes it easier to refer to the individual terms this way. The code you'll run on the CD is not broken up in this manner.

The first term *ht1* sets the rest heights for the five levels in the static diamond, and those occur at 1, 0.8, 0.5, -0.5 and -1 (see the bottom of Figure 4-4). The *ampl1-ampl5* terms allow us to individually control the amount of up and down motion for each level. Each of these are in turn controlled by a *masteramp* variable, which allows us to change the oscillational magnitude of the diamond as a whole.

The second line is a simple phased cosine wave. Five individual phase terms (*phase1-phase5*) control the relative positions of each level with respect to the others, which sets the angle (and thus the frame) where each layer reaches its maximum height. We set the two outermost levels (1 and 5) to oscillate the most, and the largest ring (3) to oscillate the least, using the *ampl1-ampl5* terms. The relative amplitude of this sinusoidal term with respect to the next exponential term is controlled by the variable *wave*.

The third line embeds a cosine wave inside an exponential term, generating a periodic bump. It is phased as well, both by a global phase term (*phase*), which sets where on the waveform the bump occurs, and individual phase terms that offset the motions of each level (*phase1-phase5*). The variable *bump* controls its overall magnitude.

We can individually adjust the amount of periodicity or bumpiness by setting the two variables *wave* and *bump*, and the overall amount of oscillation by the masteramp term that controls *ampl1-ampl5*. Figure 4-5 illustrates the motion of the five levels over 360°.

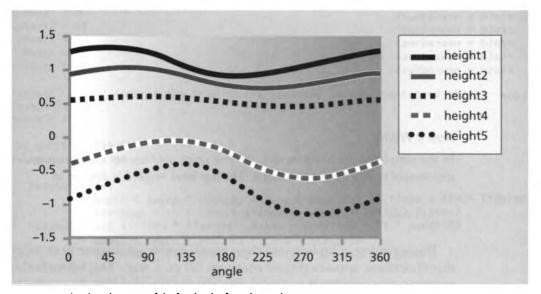


Figure 4-5 The phased motions of the five levels of our diamond

In making our diamond swim, we not only make the levels move up and down, we make the radii get larger and smaller. The resting radii are specified by *radius1-radius5*, which are then modified using various amounts of the *height* terms, such as:

newradius2 = radius2 * height2 * 1.3

This makes the vertices both go up and down and expand and shrink in diameter, in an attempt to mimic the diamond pulling in water, expanding, then thrusting it out and shrinking.

Polyray Data File

Normally with a model of this complexity, external animation with a collection of pre-calculated data files would be the only way to go. However, a diamond is such a remarkably symmetrical structure that we can exploit this symmetry to generate the diamond entirely inside Polyray. We use the motion we derived in the preceding section to control the heights of the five levels of our swimming diamond. Consider Figure 4-6, which shows a view of a cut diamond from above.

Starting from the center out, the eight inner triangles forming the octagon are coplanar, and are placed at 45° intervals around a circle. The next eight triangles out are actually tilted back into the paper, (difficult to show here), with their outermost vertices rotated 22.5° away from their base vertices.

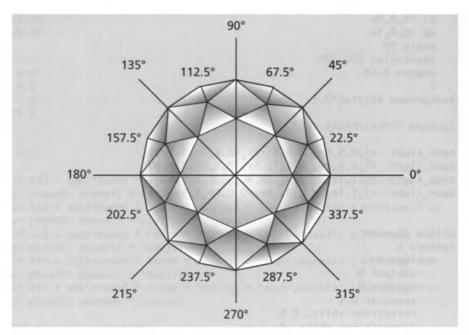


Figure 4-6 The principle controlling angles of a diamond

Every layer of control vertices is either spaced at 45° intervals around a circle, or angled 22.5° (1/16) away from that. There are five sets of eight triangles that define the top or "crown," and four sets of triangles that define the bottom, or "pavillion."

Note that the diamond has 16 sets of coplanar triangles that form trapezoids. There are eight of these on the top and eight on the bottom. The following code generates the diamond. The code is lengthy, and has been abbreviated here in the following listing. See the file DIAM1.PI for the complete listing.

```
//DIAM1.PI
start_frame 0
end_frame 71
total_frames 72
define ang frame*360/total_frames
outfile "diam1"
viewpoint {
   from <0,-8,4>
```

continued on next page

CHAPTER FOUR

```
continued from previous page
   at <0.0.0>
   up <0,0,1>
   angle 30
   resolution 320,200
   aspect 1.43
background SkyBlue*0.8
include "\PLY\COLORS.INC"
spot_light <1,0.5,0>, <-5,5, 5>,<0,4,0>,3,5,20
spot_light <1,0.5,0>, < 5,5, 5>,<0,4,0>,3,5,20
spot_light <1,1,1>, < 0,0, 5>,<0,0,0>,3,5,20
spot_light <1,1,1>, < 0,0, 5>,<0,0,0>,3,5,20
define diamond
texture {
   surface {
      ambient O
      diffuse 0
      specular 0.1
      reflection white, 0.0
      transmission white, 1, 2.6
      }
   }
object { disc <0, 0,-2.5>, <0, 0, 1>, 8 reflective_coral }
define pi 3.14159
define rad pi / 180
define radius1 1
define radius2 1.4
define radius3 1.8
define radius4 0.8
define radius5 0.01
define masteramp 1.3
define amplitude1 0.03 * masteramp
define amplitude2 0.02 * masteramp
define amplitude3 0.01 * masteramp
define amplitude4 0.04 * masteramp
define amplitude5 0.05 * masteramp
define phase 80 // hump phase
define phase1 0
define phase2 25
define phase3 50
```

```
define phase4 75
define phase5 87
define ht1 1.0
define ht2 0.8
define ht3 0.5
define ht4 -1.0
define ht5 -1.9
define wave 4
define bump 1.3
define height1 ht1 + amplitude1 * (wave * cos(rad * (ang- phase1)) + EXP(bump * ←
(1 + cos((ang- phase0- phase1) * rad))))
define height2 ht2 + amplitude2 * (wave * cos(rad * (ang- phase2)) + EXP(bump * ←
(1 + cos((ang- phase0- phase2) * rad))))
define height3 ht3 + amplitude3 * (wave * cos(rad * (ang- phase3)) + EXP(bump * ←
(1 + cos((ang- phase0- phase3) * rad))))
define height4 ht4 + amplitude4 * (wave * cos(rad * (ang- phase4)) + EXP(bump * ←
(1 + cos((ang- phase0- phase4) * rad))))
define height5 ht5 + amplitude5 * (wave * cos(rad * (ang- phase5)) + EXP(bump * ←
(1 + cos((ang- phase0- phase5) * rad))))
define newradius1 radius1*height1
define newradius2 radius2*height2*1.3
define newradius3 radius3*height3*2
define newradius4 radius4*(1.9+height4)
define newradius5 radius5
define f 1
define a 360 * f / 8
define bend1 0
define vertex01x newradius1 * cos(rad * a)
define vertexO1y newradius1 * SIN(rad * a)
define vertex01z height1
define vertex09x newradius2 * cos(rad * (a + 22.5))
define vertex09y newradius2 * SIN(rad * (a + 22.5))
define vertex09z height2
define vertex17x newradius3 * cos(rad * (a + 45))
define vertex17y newradius3 * SIN(rad * (a + 45))
define vertex17z height3
define vertex25x newradius3 * cos(rad * (a + 67.5))
define vertex25y newradius3 * SIN(rad * (a + 67.5))
define vertex25z height3
define vertex33x newradius4 * cos(rad * (a + 67.5))
define vertex33y newradius4 * SIN(rad * (a + 67.5))
                                                                   continued on next page
```

```
continued from previous page
define vertex33z height4
define f 2
define a 360 * f / 8
define bend2 45
define vertex02x newradius1 * cos(rad * a)
define vertexO2y newradius1 * SIN(rad * a)
define vertex02z height1
define vertex10x newradius2 * cos(rad * (a + 22.5))
define vertex10y newradius2 * SIN(rad * (a + 22.5))
define vertex10z height2
define vertex18x newradius3 * cos(rad * (a + 45))
define vertex18y newradius3 * SIN(rad * (a + 45))
define vertex18z height3
define vertex26x newradius3 * cos(rad * (a + 67.5))
define vertex26y newradius3 * SIN(rad * (a + 67.5))
define vertex26z height3
define vertex34x newradius4 * cos(rad * (a + 67.5))
define vertex34y newradius4 * SIN(rad * (a + 67.5))
define vertex34z height4
```

// the triangular faces listed clockwise by vertex

DIAM1.PI continues in this fashion until it becomes necessary to define the separate faces of the image. This is handled as shown in this exerpt from DIAM1.PI.

```
define diam
  object {
// Crown

  object { polygon 3,<vertex01x,vertex01y,vertex01z>,<vertex42x,vertex42y,
  vertex42z>,<vertex02x,vertex02z,vertex02z>}
  + object { polygon 3,<vertex02x,vertex02y,vertex02z>,<vertex42x,vertex42y,
  vertex42z>,<vertex03x,vertex03y,vertex03z>}
  + object { polygon 3,<vertex03x,vertex03y,vertex03z>,<vertex42x,vertex42y,
  vertex42z>,<vertex04x,vertex04y,vertex04z>}
  + object { polygon 3,<vertex04x,vertex04y,vertex04z>,<vertex42x,vertex42y,
  vertex42z>,<vertex05x,vertex05y,vertex05z>}
  + object { polygon 3,<vertex05x,vertex05y,vertex05z>,<vertex42x,vertex42y,
  vertex42z>,<vertex05x,vertex05y,vertex05z>,<vertex42x,vertex42y,
  vertex42z>,<vertex06x,vertex06y,vertex06z>}
```

```
+ object { polygon 3,<vertex06x,vertex06y,vertex06z>,<vertex42x,vertex42y,←
vertex42z>,<vertex07x,vertex07y,vertex07z>}
 + object { polygon 3,<vertex07x,vertex07y,vertex07z>,<vertex42x,vertex42y,←
vertex42z>,<vertex08x,vertex08y,vertex08z>}
 + object { polygon 3,<vertex08x,vertex08y,vertex08z>,<vertex42x,vertex42y,←
vertex42z>,<vertex01x,vertex01y,vertex01z>}
 + object { polygon 3,<vertex01x,vertex01y,vertex01z>,<vertex02x,vertex02y,←
vertex02z>,<vertex09x,vertex09y,vertex09z>}
 + object { polygon 3,<vertex02x,vertex02y,vertex02z>,<vertex03x,vertex03y, ←
vertex03z>,<vertex10x,vertex10y,vertex10z>}
 + object { polygon 3,<vertex03x,vertex03y,vertex03z>,<vertex04x,vertex04y,←
vertex04z>,<vertex11x,vertex11y,vertex11z>}
 + object { polygon 3,<vertex04x,vertex04y,vertex04z>,<vertex05x,vertex05y,←
vertex05z>,<vertex12x,vertex12y,vertex12z>}
 + object { polygon 3,<vertex05x,vertex05y,vertex05z>,<vertex06x,vertex06y,←
vertex06z>,<vertex13x,vertex13y,vertex13z>}
 + object { polygon 3,<vertex06x,vertex06y,vertex06z>,<vertex07x,vertex07y,←
vertex07z>,<vertex14x,vertex14y,vertex14z>}
 + object { polygon 3,<vertex07x,vertex07y,vertex07z>,<vertex08x,vertex08y,←
vertex08z>,<vertex15x,vertex15y,vertex15z>}
 + object { polygon 3,<vertex08x,vertex08y,vertex08z>,<vertex01x,vertex01y,←
vertex01z>,<vertex16x,vertex16y,vertex16z>}
diam {rotate <0,0,ang/8>
      translate <0,0,-1.5+height1>}
```

The motion has already been covered in the QuickBasic simulation code. It translates without a hitch. Keeping track of all the vertices may be tedious, but number the vertices in Figure 4-4, starting with the top octagon 1 through 8, then 9 through 16 for the next ring, and so on up to 40, then number the bottom point 41 and the center of the top octagon 42, and you have it. This allows you to keep track of the 42 control vertices. Then it becomes just a matter of calling up each triangle in clockwise order from the outside, although providing you're consistent, (clockwise or counterclockwise, pick one), it doesn't matter. A sample image is shown if Figure 4-7.

Comments

Changing the ratio of the magnitude of *bump* to *wave* produces a range of motions, from a rubber spring-like up-and-down oscillation to a periodic pulse. The exponent in the *bump* term limits the acceptable values for *bump* to a maximum of about 1.3, although watching it go non-linear can also be amusing.

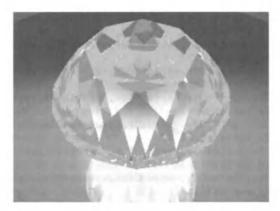


Figure 4-7 A girl's best friend



Figure 4-8 Diamond in a watery setting

The previous background with simple yellow backlighting code produces a pastel blue diamond. A background providing more saturated colors uses the definitions featured in DIAM2.PI, but the rest of the program remains essentially the same. Due to the extra lights and textures, it takes considerably longer to render. But, as seen in Figure 4-8, it's worth the wait.

```
//DIAM2.PI
```

start_frame 0
end_frame 71
total_frames 72

define ang frame*360/total_frames

```
outfile "diam2"
viewpoint {
   from <0,-5,5>
   at <0.0.0>
   up <0,0,1>
   angle 50
   resolution 320,240
   aspect 1.33
background sky_blue*0.16
include "\PLY\COLORS.INC"
spot_light <1,0.5,0>, <-5,5, 5>,<0,4,0>,3,5,20
spot_light <1,0.5,0>, < 5,5, 5>,<0,4,0>,3,5,20
spot_light <1,1,1>*0.5, < 0,0, 5>,<0,0,0>,3,5,20
spot_light <1,1,1>*0.5, < 0,0, 5>,<0,0,0>,3,5,20
light white *6, <-50, -50, 50>
light white *6, < 50, -50, 50>
light white, <30,-100,100>
light white, <-30,-100,100>
define diamond
texture {
   surface {
      ambient O
      diffuse 0
      specular 0.0
      reflection white, 0.1
      transmission white, 1, 2.6
      }
   }
object { disc <0, 0,-2.5>, <0, 0, 1>, 16 reflective_coral }
define pi 3.14159
define rad pi/180
define blue_ripple
   texture {
      noise surface {
         color navyblue
         normal 2
         frequency 1
         phase -frame*rad*10
         bump_scale 2
         ambient 0.2
         diffuse 0.4
         specular yellow, 0.5
```

continued on next page

The two high-intensity light sources, six times maximum white, are added as shown here:

```
light white*6, <30,-100,100> light white*6, <-30,-100,100>
```

These are perfectly legal, and actually necessary to make the background—dark, reflective surface—visible. The intense lights also add a sparkle to the diamond.



4.3 How do I...

Turn a Moravian star inside out?

You'll find the code for this in: PLY\CHAPTER4\MORAVIAN

Problem

Complex polyhedra like snowflakes, zeolite (a cage-like structure found in clays) and Moravian stars are crystalline entities that exhibit various collective symmetries. These can be used to create some interesting animations. The vertices controlling their shape may be grouped into logical sets, i.e. inner and outer points, or more simply: peaks and valleys. A somewhat simplified Moravian star (see Figure 4-9) is one such figure. It makes an ideal candidate for collective radial translations of these control sets which can be thought of as "breathing." We convert the peaks to valleys and the valleys to peaks, turning the figure inside-out, and rotate the figure to show off the motion.

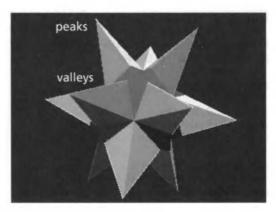


Figure 4-9 Peaks and valleys in a simple Moravian star

Technique

Jay Schumacher did a very nice Moravian star-type figure illuminated all around by various colored spots. These lights make it possible to define the figure as *matte_white* and still end up with a colorful figure. Figure 4-9 shows how this figure breaks down naturally into peaks and valleys. This animation moves these two sets of points in and out at different times to create some interesting in-between geometry, and tumbles the shape as we do it.

This shape's six-way symmetry allows us to define one of the sides, then rotate that definition five times to form the complete figure.

Steps

The animation STAR.PI is so simple that it has no QuickBasic simulation code. We define the geometry, breathe life into it, and set it tumbling in the following script:

```
// STAR.PI
// Star That Tumbles
// Original Geometry - Jay Schumacher
start_frame 0
end_frame 179
total_frames 180
define index frame*360/total_frames
outfile "star"
define pi 3.14159
define rad pi/180
```

continued on next page

CHAPTER FOUR

```
continued from previous page
include "\PLY\COLORS.INC"
// set up background color & lights
background midnightblue
define dim 1
light <1,0,0>*dim,< 10, 0, 0>
light <1,1,1>*dim,< 0, 10, 0>
light <0,0,1>*dim,< 0, 0, 10>
light <1,1,0>*dim,<-10, 0, 0>
light <0,1,1>*dim,< 0,-10, 0>
light <1,0,1>*dim,<0,0,-10>
viewpoint {
  from <3, 6, 4>
   at <0,0,0>
  up <0, 1, 0>
   angle 30
   resolution 320,200
   aspect 1.43
   }
// the phased breathing terms
define ph1 1+0.5*cos(index*rad) // goes from 0.5 to 1.5
define ph2 0.5*(1+0.5*sin(index*rad)) // goes from 0.25 to 0.75
// valleys ("F"ace Centers)
define f01 <0,1,1>*ph1
define f02 <1,0,1>*ph1
define f03 <0,-1,1>*ph1
define f04 <-1,0,1>*ph1
// peaks ("T"ips)
define t01 <1,1,1>*ph2
define t02 <1,-1,1>*ph2
define t03 < -1, -1, 1 > *ph2
define t04 <-1,1,1>*ph2
// "C"entral point
define c01 < 0.0,1 >
define face
  object {
     object { polygon 3, f01,t01,c01 matte_white}
    + object { polygon 3, t01,f02,c01 matte_white}
    + object { polygon 3, f02,t02,c01 matte_white}
    + object { polygon 3, t02,f03,c01 matte_white}
    + object { polygon 3, f03,t03,c01 matte_white}
```

```
+ object { polygon 3, t03,f04,c01 matte_white}
+ object { polygon 3, f04,t04,c01 matte_white}
+ object { polygon 3, t04,f01,c01 matte_white}
}

define star
  object {
    face
+ face { rotate <0, 90,0> }
+ face { rotate <0,180,0> }
+ face { rotate <0,270,0> }
+ face { rotate <90,0,0> }
+ face { rotate <270,0,0> }
}
star {rotate <index,45*sin(2*index*rad),90*cos(index*rad)>}
```

How It Works

There were two classes of points, peaks, or tips (t) and valleys or face centers (f and c). The motions of the centers and tips are phased 90° apart, and ph1 has twice the magnitude of ph2, making it easy to tell the difference between the two as each set takes center stage. The figure expands to a star, contracts to a ball, extrudes into connected plates, and then returns to the star shape (see Figure 4-10).

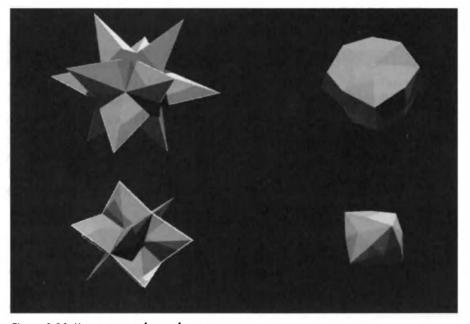


Figure 4-10 Moravian star in four configurations

Comments

This entire figure is defined by eight vertices and one central point. Simple rotations of these points can result in all sorts of interesting shape changes. To rotate the tips:

```
// peaks ("T"ips)
define t01 rotate(<1,1,1>*ph2, <0,0,45*sin(3*index*rad)>)
define t02 rotate(<1,-1,1>*ph2,<0,0,45*sin(3*index*rad)>)
define t03 rotate(<-1,-1,1>*ph2,<0,0,45*sin(3*index*rad)>)
define t04 rotate(<-1,1,1>*ph2,<0,0,45*sin(3*index*rad)>)
```

We can also change the phase (relative timing) of the in-and-out motions for members of each symmetrical set, but due to pre-existing symmetry constraints, the figure will no longer be solid. Cracks will appear between the segments. This isn't necessarily a bad thing, though.

While this is a simple example of triangle mesh morphing, the same principle can be applied to far more complex objects. You can purchase triangle and polygon mesh models for things like tennis shoes and busts of Beethoven. The vertices can be moved using functions based on their distance from strategically placed control points, and morphed into all sorts of interesting shapes.



4.4 How do I...

Morph functionally defined objects?

You'll find the code for this in: PLY\CHAPTER4\CARB

Problem

In the previous animation (STAR.PI), we dealt with the periodic displacements of the discrete vertices controlling the shape of a solid object defined as a triangle mesh. It's also possible to take a functionally defined object and modify the function to do the same type of thing. Let's take the cubical object from the Disco Inferno animation (Section 3.3) and blow out its sides sequentially as we rotate it.

Technique

Functionally defined objects get their shapes from solving the equations that define their surfaces. Such is the case with the tooth-like object we used in Section 3.3, based on an object from an article by Don Mitchell and Pat Hanrahan in the 1992 SIGGRAPH Proceedings on the illumination of curved

reflectors. The object is defined where points on its surface solve the relationship:

$$x^4 + y^4 + z^4 - x^2 - y^2 - z^2 = 0$$

For example, the points <1,1,1>, <-1,-1,-1> and in fact the other six corners of a unit cube all satisfy this equation. The in-between points are what's hard to see what's going on intuitively. Polyray gets its name from its ability to ray trace objects defined by polynomials, so this is right up its alley.

In order to change the shape of this object, we need to add additional factors indexed by the frame count that move this surface around. We do so by multiplying x, y, and z by three new variables a, b, and c. In code this is handled:

```
define m 360/total_frames
define off sin(frame*m*8*rad)/2

define a (sin(frame*m*rad)+off)/2
define b (sin((frame*m+120)*rad)+off)/2
define c (sin((frame*m+240)*rad)+off)/2
...
polynomial a*x^4 + b*y^4 + c*z^4 - a*x^2 - b*y^2 - c*z^2
```

These new variables shift the solution for the surface so that it opens up revealing interesting interior details that resemble a four barrel carburetor—so we've named the device CARB (see Figure 4-11).

Polyray Code

CARB is a multi-ported, topologically varied object. (It has a variable number of holes in it.) Simulating functionally defined surfaces inside QuickBasic is extremely difficult. It requires an accelerated solution to find



Figure 4-11 Our carburetor

code like Polyray's marching cubes algorithm; this isn't exactly "quick-n-dirty" like most of the QuickBasic code we've written so far. We'll forgo the simulation and cut straight to Polyray code, CARB.PI, since that's the best way to do this.

```
// CARB.PI - Hyperdimensional Hyperactive Carburetor
// Polyray input file: Jeff Bowermaster
// set up the frames
start_frame 0
end frame 359
total_frames 360
outfile "carb"
define pi 3.14159
define rad pi/180
define ang frame*360/total_frames
define off sin(ang*8*rad)/2
define a (sin(ang*rad)+off)/2
define b (sin((ang+120)*rad)+off)/2
define c (sin((ang+240)*rad)+off)/2
// camera
viewpoint {
   from <5, 3, 5>
        <0,0,0>
        <0,1,0>
   angle 30
   resolution 160,120
   aspect 1.333
// get various surface finishes
include "\PLY\COLORS.INC"
// Set up background color & lights
background black
light < 3, 0, 0>
light < 0, 3, 0 >
light < 0, 0, 3>
// carburetor object
object {
   object {
      polynomial a*x^4 + b*y^4 + c*z^4 - a*x^2 - b*y^2 - c*z^2
      root_solver Ferrari
   & object { box <-2, -2, -2>, <2, 2, 2> }
```

```
rotate <0, ang, 0>
   reflective_coral
// create a ground plain
object {
   object {
     polynomial y + 0.01
      root_solver Ferrari
   & object { box <-4, -0.5, -4>, <4, 0.5, 4> }
   translate < 0.0, -2.0, 0.0 >
   texture { checker matte_white, matte_black }
   scale <0.5, 0.5, 0.5>
   }
// create a sky plain
object {
   object {
      polynomial y + 0.01
      root_solver Ferrari
   & object { box <-4, -0.5, -4>, <4, 0.5, 4> }
   translate < 0.0, 4.0, 0.0 >
   texture { checker matte_white, matte_black }
   scale <1, 1, 1>
   }
// create a left plain
object {
   object {
      polynomial z + 0.01
      root_solver Ferrari
   & object { box <-4, -4, -0.5>, <4, 4, 0.5> }
   translate < 0.0, 0.0, 5.5 >
   texture { checker matte_white, matte_black }
   scale <1, 1, 1>
// create a right plain
object {
   object {
      polynomial x + 0.01
      root_solver Ferrari
   & object { box <-0.5, -4, -4>, <0.5, 4, 4> }
   translate < 5.5, 0.0, 0.0 >
```

continued on next page

```
continued from previous page
  texture { checker matte_white, matte_black }
  scale <1, 1, 1>
}
```

How It Works

Repeating the function modifications here again, the basic cubic surface, represented by

```
x^4 + y^4 + z^4 - x^2 - y^2 - z^2 = 0
```

is modified using three phased coefficients a, b, and c

```
define off sin(ang*8*rad)/2
define a (sin(ang*rad)+off)/2
define b (sin((ang+120)*rad)+off)/2
define c (sin((ang+240)*rad)+off)/2
```

The additional variables move the solution out from the cube and, using a periodic function like a sine, does so rhythmically. The sine terms are themselves further modified by another term *off*, running eight times the loop speed, making the Carb object quiver and shimmy as it slowly rotates.

The figure was clipped by a "pizza box," a wide thin box, eight by eight with a height of one. It's defined as

```
& object { box <-4, -0.5, -4>, <4, 0.5, 4> }
```

to allow us to see the inner solution of the surface without the outer layers getting in the way.

The surface of CARB is a reflecting gold-like texture called *reflective_coral*. It's difficult to tell what's going on with the shifting surface morphology without some kind of recognizable reference texture. For this, we place a number of black and white checkerboards (ground, sky, left and right planes) around it, and their reflection on the surface shows what's happening with the curvature of our carburetor.

A syntax consistent used with the rest of the animation seems a little brutal now, but it solves a first-order equation for a surface, then clips it using the "pizza box" routine. Here's the code that establishes this:

```
// create a ground plain
object {
  object {
    polynomial y + 0.01
      root_solver Ferrari
    }
  & object { box <-4, -0.5, -4>, <4, 0.5, 4> }

  translate < 0.0,-2.0,0.0>
  texture { checker matte_white, matte_black }
  scale <0.5, 0.5, 0.5>
}
```

The same plane can be just as easily defined using a polygon:

```
object {
   polygon 4, <-4,-1.99,-4>,< 4,-1.99,-4>,<4,-1.99,4>,<-4,-1.99,4>
   texture { checker matte_white, matte_black }
   scale <0.5, 0.5, 0.5>
}
```



4.5 How do I...

Make a manta wave motion with a flapping wedge?

You'll find the code for this in: PLY\CHAPTER4\MANTA

Problem

Autodesk 3D Studio has a series of add-on IPAS routines (short for Image processing, Procedural modeling, Animation stand-in, and Solid pattern external processes) with some wonderful animation effects. Unfortunately, they're only for use inside 3D Studio. They generate effects, such as volcanoes, fountains, and fireworks. There are kits you can use to write your own, so many third party routines are now available. Access to 3D Studio and the software necessary to compile IPAS routines will run you several thousand dollars—beyond the reach of most hobbyists. However, looking carefully at one routine in particular, the Manta sample animation file, it looks like it's just an expanding sine function mapped on the edges of a wedge. It should be possible to duplicate this effect with a functional heightfield inside Polyray.

Technique

Figure 4-12 shows a single frame from the Yost Group's IPAS sample file, and provides an example of our goal. (Incidentally, the Yost Group developed 3D Studio for Autodesk.) The animation shows a ribbon that flaps with a very pleasing fluid motion.

Note the expansion of the deformations down the length of the wedge. It reaches its greatest amplitude at the edges. This effect is generated by functionally warping a surface with a sine wave as the viewpoint moves away from the center line. The combination of these expanding deformations with a cone clipping this surface increases the amplitude as we move out and away from the origin for this figure. This is the gist of MWAVE.PI, which is written as follows.

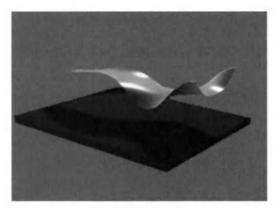


Figure 4-12 A frame from the Yost Group's Manta animation sample file

```
//MWAVE.PI - Manta Wave Height Field
start_frame 0
end frame 29
total_frames 30
outfile mw
define pi 3.14159
define time -2*pi*frame/total_frames
// set up the camera
viewpoint {
   from <10,5,18.5>
   at <0,-1.5,9.5>
   up <0,1,0>
   angle 28
   resolution 320,240
   aspect 1.33
   }
// get various surface finishes
include "\PLY\COLORS.INC"
define steel_blue
texture {
   surface {
      ambient steel_blue, 0.3
      diffuse steel_blue, 0.8
      specular white, 0.7
      }
   }
// set up background color & lights
background MidnightBlue
spot_light <0.5,0.5,0.5>, <0,7,5>, <0,-1,9>, 3, 10, 25
```

```
spot_light <0.5,0.5,0.5>, <0,10,9>, <0,-1,9>, 3, 10, 25
// defaults were g1 0.08 and g2 0.01
define g1 0.01
define g2 0.08
define HFn g1*z*sin(z+time) + g2*(z/6)^4 * sin(z+time) * sin(x)^4
define detail 40
// define a sinusoidal surface
object {
  object {
        smooth_height_fn detail, detail, -3, 3, 4, 13, HFn
        reflective_grey
   & object {
        cone <0, 0, 20>, 3, <0, 0, 0>, 0
        translate <0,0,1>
        }
    }
object {
  box <-3, -3, 4>, <3, -2.5, 13>
   steel_blue
```

How It Works

The ribbon extends down the z axis with the x axis running across the ribbon (see Figure 4-13).

The driving function for this surface is the following equation:

```
define time -2*pi*frame/total_frames define g1 0.01 define g2 0.08 define HFn (g1*z*sin(z+time)) + (g2*(z/6)^4 * sin(z+time) * sin(x)^4)
```

Although there are a lot of variables here, we really only have two terms. The first one is nothing more than a sine wave whose amplitude increases as we progress down the ribbon.

The second term has the same functionality, but we've raised the length dependency to a power to make it increase more rapidly, and added an x component to flap the ribbon across its width $(\sin(x)^4)$. We adjust the variables g1 and g2 to achieve the proper mix of these two terms, giving us control over the rate at which the flapping increases.

The $sin(x)^4$ function is flatter across the middle than first order trig functions (see Figure 4-14), and provided we don't exceed ± 0.5 radians ($\pi/2$)

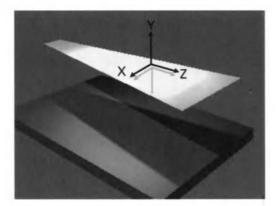


Figure 4-13 Geometry of our ribbon

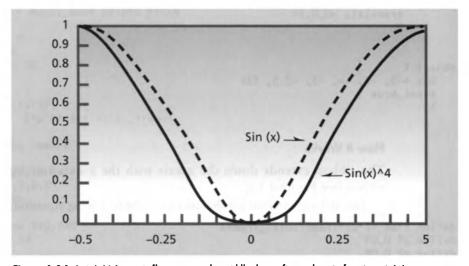


Figure 4-14 A $\sin(x)^4$ term is flatter across the middle than a first order tria function, $\sin(x)$

for our ribbon width, we'll make the edges with suitably exaggerated motion at the tips. We'll adjust the cone that clips the ribbon to keep us inside this range at the very ends of our ribbon. A view of our manta surface is shown in Figure 4-15.

Comments

There's a problem clipping a heightfield with a cone, and sometimes parts of the ribbon seen from below disappear entirely (see Figure 4-16). This means we're constrained to limit the amplitude of the oscillation and/or the viewpoint to avoid making a broken ribbon, although it's an interesting effect.

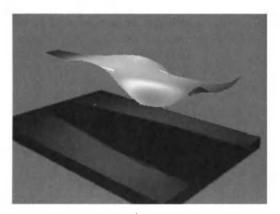


Figure 4-15 Our manta surface

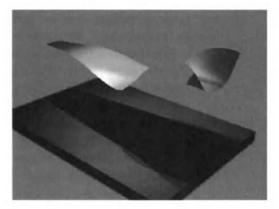


Figure 4-16 Ribbon breaking ceremony; hold the applause



🏅 4.6 How do I...

Make the letters HMM "happy-train"?

You'll find the code for this in: PLY\CHAPTER4\HMM

Problem

Solid block characters commonly used for company logos don't bounce very well. About the best you can do is stretch them along their x, y, and z axes and hope for the best. If you *really* want them to bounce, twist and wiggle, you have to break them up and give them room to dance. What we do is figuratively feed them to a bread slicer, toss out every other slice, and use the spare room to move the pieces around. The venerable IBM® logo is a perfect

example of this kind of lettering, but understandably, companies like IBM are very proud of their logos. The thought of having thousands of twisting, wiggling, raving polka dot IBM logos bouncing around every bulletin board from Armonk to Anchorage caused them some concern. So instead, we'll opt for things that make you go "HMM." These are the initials of my research director at Virginia Tech, Dr. Harold M. McNair. Those initials always made it ambiguous whether a document was something he was wondering about, or something he had written.

Technique

The classic Steam Boat Willie cartoon of the 30s had almost every object in the scene bouncing up and down in time with the music. Apparently, Willie himself must have been two-cycle, since he bounced twice as fast as he walked. We're going to animate the letters HMM to give it this type of bounce, swaying back and forth, moving past the camera in a motion known as "happy-train." We'll refrain from making it whistle, though.

First generate the letters, using QuickBasic to do the nasty low-level stuff. The procedure involves sketching out letters on graph paper and deriving the controlling vertices for the various blocks. Concentrate on finding critical points: things like corners, centers of circles (none here, but keep it in mind for your logos), things that line up, distances... and manually extract as much information as possible. Any regularities in the letters are exploited to simplify the creation of the model. In this particular case, each letter contained regularly spaced horizontal stripes, many of which line up vertically. This extracted data is placed into DATA statements and called in loops to convert it into boxes and polygons, the elements of our logo. The following code (HM.BAS) displays both the letters as they're created and writes a Polyray file defining the blocks.

Boxes suffice for some of the pieces, but for slanted sections of the *M* that aren't square, polygons must be used six at a time to construct trapezoidal enclosures. The code creates an include file called HM.INC with each piece of each letter named *H1-H8* and *M1-M8*, and displays the letters on the screen (see Figure 4-17). We needn't make two *M's*; we can translate a copy of the first *M* to make a second one.

QuickBasic Code

The following program generates the letters in Figure 4-18. It uses blocks of data statements and regularities inside the letters themselves to make the final model. Again, the code is long and has been abbreviated here. Check the file for the complete listing.

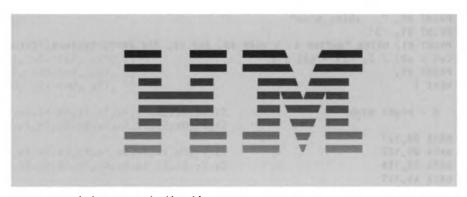


Figure 4-17 The letters HM in sliced bread font

```
' HM.BAS - Creates two block letters H and M
WINDOW SCREEN (-20, -50)-(299.5, 189.5)
OPEN "hm.inc" FOR OUTPUT AS #1
'depth
z1 = -50
z2 = 50
'H - left side
DATA 31,65
DATA 31,65
DATA 44,65
DATA 44,115
DATA 44,115
DATA 44,65
DATA 31,65
DATA 31,65
FOR i = 1 TO 8
READ x1, x2
y1 = (i - 1) * 13
y2 = (i - 1) * 13 + 7
LINE (x1, y1)-(x2, y2), 15, B
Name$ = RIGHT$("HL" + LTRIM$(STR$(i)), 3)
Center$ = RIGHT$("HLC" + LTRIM$(STR$(i)), 4)
PRINT #1, "define "; Name$
PRINT #1, "object {"
PRINT #1, USING " box <###.##, ###.##, ###.#+>, <###.##, ###.##, ###.##>"; ←
x1, y1, z1, x2, y2, z2
                                                                   continued on next page
```

CHAPTER FOUR

```
continued from previous page
            shiny_blue"
PRINT #1, "
PRINT #1, "}"
PRINT #1, USING "define \ \ <###.##, ###.##, ###.#"; Center$, (x1 + x2) / 2,←
(y1 + y2) / 2, (z1 + z2) / 2
PRINT #1,
NEXT i
' H - right side
DATA 93,127
DATA 93,127
DATA 93,115
DATA 44,115
DATA 44,115
DATA 93,115
DATA 93,127
DATA 93,127
FOR i = 1 TO 8
READ x1, x2
y1 = (i - 1) * 13
y2 = (i - 1) * 13 + 7
LINE (x1, y1)-(x2, y2), 15, B
Name\$ = RIGHT\$("HR" + LTRIM\$(STR\$(i)), 3)
Center$ = RIGHT$("HRC" + LTRIM$(STR$(i)), 4)
PRINT #1, "define "; Name$
PRINT #1, "object {"
PRINT #1, USING " box <###.##, ###.##, ###.##>, <###.##, ###.##, ###.##>"; ←
x1, y1, z1, x2, y2, z2
PRINT #1, "
            shiny_blue"
PRINT #1, "3"
PRINT #1, USING "define \ \ <###.##, ###.##, ###.#"; Center$, (x1 + x2) ←
/2, (y1 + y2) / 2, (z1 + z2) / 2
PRINT #1,
NEXT i
' M
'the top V
DATA 183,0,200,44
DATA 143,143,157
'determine edge points for line
READ a, b, c, d
slope = (c - a) / (d - b)
'PRINT slope
FOR i = 1 TO 3
   y1 = (i - 1) * 13
  y2 = (i - 1) * 13 + 7
  x1 = a + y1 * slope
  x2 = a + y2 * slope
```

```
READ xO
  LINE (x0, y1)-(x1, y1), 15
  LINE (x1, y1)-(x2, y2), 15
  LINE (x2, y2)-(x0, y2), 15
  LINE (x0, y2)-(x0, y1), 15
       x0,y1,z1,x1,y1,z1,x1,y1,z2,x0,y1,z2
'bottom x0,y1,z1,x2,y2,z1,x2,y2,z2,x0,y2,z2
'front x0,y1,z1,x1,y1,z1,x2,y2,z1,x0,y2,z1
'back
      x0,y1,z2,x1,y1,z2,x2,y2,z2,x0,y2,z2
'left
      x0,y1,z1,x0,y1,z2,x0,y2,z2,x0,y2,z1
'right x1,y1,z1,x2,y2,z1,x2,y2,z2,x1,y1,z2
Name\$ = RIGHT\$("ML" + LTRIM\$(STR\$(i)), 3)
Center$ = RIGHT$("MLC" + LTRIM$(STR$(i)), 4)
PRINT #1, "define "; Name$
PRINT #1, "object {"
PRINT #1, USING " object { polygon 4, <###.##, ###.##, ###.##>, <###.##, ←
###.##, ###.##>,<###.##> } +"; x0, ←
y1, z1, x1, y1, z1, x1, y1, z2, x0, y1, z2
PRINT #1, USING " object { polygon 4, <###.##, ###.##, ###.##>, <###.##, ←
###.##, ###.##>,<###.##> } +"; x0, ←
y2, z1, x2, y2, z1, x2, y2, z2, x0, y2, z2
PRINT #1, USING " object { polygon 4, <###.##, ###.##, ###.##>, <###.##, ←
###.##, ###.##>,<###.##> } +"; x0, ←
y1, z1, x1, y1, z1, x2, y2, z1, x0, y2, z1
PRINT #1, USING " object { polygon 4, <###.##, ###.##, ###.##>, <###.##, ←
###.##, ###.##>,<###.##> } +"; x0, ←
y1, z2, x1, y1, z2, x2, y2, z2, x0, y2, z2
PRINT #1, USING " object { polygon 4, <###.##, ###.##, ###.#*>, <###.##, ←
###.##, ###.##>,<###.##, ###.##>,<###.##>} +"; xO, ←
y1, z1, x0, y1, z2, x0, y2, z2, x0, y2, z1
PRINT #1, USING " object { polygon 4, <###.##, ###.##, ###.##>, <###.##, ←
###.##, ###.##>,<###.##, ###.##>,<###.##>} "; x1, ←
y1, z1, x2, y2, z1, x2, y2, z2, x1, y1, z2
PRINT #1, " shiny_blue"
PRINT #1, " }"
PRINT #1, USING "define \ \ <###.##, ###.##>"; Center$, (x0 + (x1 + \leftarrow
x2) / 2) / 2, (y1 + y2) / 2, (z1 + z2) / 2
PRINT #1,
NEXT i
CLOSE #1
```

How It Works

There's a lot here, but basically each bar in the logo is six units high and they are spaced seven units apart. Specify the left and right x coordinates for each block and loops handle the rest of the details for the H. The M is more involved. The left and right vertical pillars are easy, but the diagonal mid-section, shaped like a V, requires solving the equation for a line in order to set the edge position at the various levels. Each block is constructed out of polygon patches. Use heavy formatting to generate a readable Polyray data file, a section of which follows:

```
define HL1
object {
  box < 31.00,
                 0.00, -50.00, < 65.00, 7.00, 50.00
  shiny_blue
}
define HLC1 < 48.00, 3.50,
                             0.00>
define HL2
object {
  box < 31.00, 13.00, -50.00>, < 65.00, 20.00, 50.00>
  shiny_blue
}
define HLC2 < 48.00, 16.50,
                             0.00>
define HL3
object {
  box < 44.00, 26.00, -50.00>, < 65.00, 33.00, 50.00>
  shiny_blue
define HLC3 < 54.50, 29.50,
                             0.00>
define HL4
  box < 44.00, 39.00, -50.00>, <115.00, 46.00, 50.00>
  shiny_blue
define HLC4 < 79.50, 42.50,
                             0.00>
. . .
define ML1
object {
  object { polygon 4, <143.00, 0.00, -50.00>, <183.00,
                                                          0.00, ←
-50.00>,<183.00, 0.00, 50.00>,<143.00,
                                          0.00, 50.00 >  +
  object { polygon 4, <143.00,
                               7.00, -50.00>, <185.70,
                                                          7.00, ←
                 7.00, 50.00>,<143.00,
                                          7.00, 50.00> } +
-50.00>,<185.70,
                               0.00, -50.00>, <183.00,
  object { polygon 4, <143.00,
                                                          0.00, ⇐
-50.00>,<185.70,
                 7.00, -50.00>,<143.00, 7.00, -50.00> } +
  object { polygon 4, <143.00, 0.00, 50.00>, <183.00,
                                                          0.00, ←
50.00>,<185.70, 7.00, 50.00>,<143.00, 7.00, 50.00> } +
```

```
object { polygon 4, <143.00, 0.00, -50.00>, <143.00,
                                                           0.00, ←
                 7.00, 50.00>,<143.00, 7.00, -50.00> } +
50.00>,<143.00,
   object { polygon 4, <183.00, 0.00, -50.00>, <185.70,
                                                           7.00, <del>=</del>
-50.00>,<185.70,
                 7.00, 50.00>,<183.00, 0.00, 50.00> }
   shiny_blue
define MLC1 <163.68, 3.50,
                              0.00>
define ML2
object {
   object { polygon 4, <143.00, -50.00>, <188.02, 13.00, \leftarrow
-50.00>,<188.02, 13.00, 50.00>,<143.00, 13.00, 50.00> } +
   object { polygon 4, <143.00, 20.00, -50.00>, <190.73, 20.00, ←
-50.00>,<190.73, 20.00, 50.00>,<143.00, 20.00, 50.00> } +
   object { polygon 4, <143.00, -50.00>, <188.02, 13.00, \leftarrow
-50.00>,<190.73, 20.00, -50.00>,<143.00, 20.00, -50.00> } +
  object { polygon 4, <143.00, 13.00, 50.00>, <188.02, 13.00, \Leftarrow
50.00>,<190.73, 20.00, 50.00>,<143.00, 20.00, 50.00> } +
   object { polygon 4, <143.00, 13.00, -50.00>, <143.00, 13.00, \leftarrow
50.00>,<143.00, 20.00, 50.00>,<143.00, 20.00, -50.00> } +
   object { polygon 4, <188.02, 13.00, -50.00>, <190.73, 20.00, \Leftarrow
-50.00>,<190.73, 20.00, 50.00>,<188.02, 13.00, 50.00> }
   shiny_blue
define MLC2 <166.19, 16.50,
                             0.00>
define ML3
object {
  object { polygon 4, <157.00, 26.00, -50.00>, <193.05, 26.00, ←
-50.00>,<193.05, 26.00, 50.00>,<157.00, 26.00, 50.00> } +
  object { polygon 4, <157.00, 33.00, -50.00>, <195.75, 33.00, ←
-50.00>,<195.75, 33.00, 50.00>,<157.00, 33.00, 50.00> } +
   object { polygon 4, <157.00, 26.00, -50.00>, <193.05, 26.00, \Leftarrow
-50.00>,<195.75, 33.00, -50.00>,<157.00, 33.00, -50.00> } +
   object { polygon 4, <157.00, 26.00, 50.00>, <193.05, 26.00, ←
50.00>,<195.75, 33.00, 50.00>,<157.00, 33.00, 50.00> } +
   object { polygon 4, <157.00, 26.00, -50.00>, <157.00, 26.00, \Leftarrow
50.00>,<157.00, 33.00, 50.00>,<157.00, 33.00, -50.00> } +
  object { polygon 4, <193.05, 26.00, -50.00>, <195.75, 33.00, \Leftarrow
-50.00>,<195.75, 33.00, 50.00>,<193.05, 26.00, 50.00> }
  shiny_blue
define MLC3 <175.70, 29.50,
                              0.00>
. . .
(see the file HM.INC for more details)
```

Oops

Screen coordinates are normally defined using the lower-left corner of the screen as (0,0); however, for some perverse reason, while collecting the vertex data for this model, the *upper*-left corner of the screen was assigned

the coordinates (0,0). Positive values flood the rest of the screen. This means that letters are upside down. The final code should make them sway so the sections closest to the axes remain steady and sections farther out sway. In other words, these letters will sway like kids on monkey bars (top stable/bottom swaying) rather than an overloaded truck (bottom stable/top swaying), which is what we're after. Rather than recollect the data, we can rectify this situation by flipping the normal "up" vector from <0,1,0> to <0,-1,0> and call the blocks defining each slice of the letters in reverse order (H8 -> H1 rather than H1 -> H8), as shown:

```
define H8 object { HL1 + HR1 }
define H7 object { HL2 + HR2 }
define H6 object { HL3 + HR3 }
define H5 object { HL4 }
define H4 object { HL5 }
define H3 object { HL6 + HR6 }
define H2 object { HL7 + HR7 }
define H1 object { HL8 + HR8 }
```

Polyray Data File

The following Polyray data file (HMM.PI) uses the geometry in the *include* file HM.INC, built from the QuickBasic code file, to construct our happy-train HMM logo.

```
// HMM.PI - Happy-Train Letters
start frame O
end_frame 162
total_frames 163
outfile "HMM"
viewpoint {
   from <55,25,300>
   at <125,55,0>
   up <0,-1,0>
   angle 30
   resolution 160,120
   aspect 1.333
include "\PLY\COLORS.INC"
define Ochre <0.6,0.3,0.0>
background Ochre
light white, < -50, -100, 250>
light white, < 300,-100,250>
include "hm.inc"
```

```
define H8 object { HL1 + HR1 }
define H7 object { HL2 + HR2 }
define H6 object { HL3 + HR3 }
define H5 object { HL4 }
define H4 object { HL5 }
define H3 object { HL6 + HR6 }
define H2 object { HL7 + HR7 }
define H1 object { HL8 + HR8 }
define M8 object { ML1 + MR1 }
define M7 object { ML2 + MR2 }
define M6 object { ML3 + MR3 }
define M5 object { ML4 + MML4 + MMR4 + MR4 }
define M4 object { ML5 + MM5 + MR5 }
define M3 object { ML6 + MM6 + MR6 }
define M2 object { ML7 + MM7 + MR7 }
define M1 object { ML8 + MM8 + MR8 }
define M28 object { ML1 + MR1 translate <131,0,0>}
define M27 object { ML2 + MR2 translate <131,0,0>}
define M26 object { ML3 + MR3 translate <131,0,0>}
define M25 object { ML4 + MML4 + MMR4 + MR4 translate <131,0,0>}
define M24 object { ML5 + MM5 + MR5 translate <131,0,0>}
define M23 object { ML6 + MM6 + MR6 translate <131,0,0>}
define M22 object { ML7 + MM7 + MR7 translate <131,0,0>}
define M21 object { ML8 + MM8 + MR8 translate <131,0,0>}
define major 0.2
define minor 0.05
define spacing 13
define pi 3.14159
define rad pi/180
define Y1 O* spacing * major * sin(frame*24*rad)
define Y2 1* spacing * major * sin(frame*24*rad)
define Y3 2* spacing * major * sin(frame*24*rad)
define Y4 3* spacing * major * sin(frame*24*rad)
define Y5 4* spacing * major * sin(frame*24*rad)
define Y6 5* spacing * major * sin(frame*24*rad)
define Y7 6* spacing * major * sin(frame*24*rad)
define Y8 7* spacing * major * sin(frame*24*rad)
define amp 2.5
define xr1 0 * amp * sin(rad * frame * 6)
define xr2 1 * amp * sin(rad * frame * 6)
define xr3 2 * amp * sin(rad * frame * 6)
define xr4 3 * amp * sin(rad * frame * 6)
define xr5 4 * amp * sin(rad * frame * 6)
define xr6 5 * amp * sin(rad * frame * 6)
define xr7 6 * amp * sin(rad * frame * 6)
```

continued on next page

```
continued from previous page
define xr8 7 * amp * sin(rad * frame * 6)
define train 270-frame*4
H1 {translate <train, Y1, 0> rotate <xr1,0,0> }
H2 {translate <train, Y2, 0> rotate <xr2,0,0> }
H3 {translate <train, Y3, 0> rotate <xr3,0,0> }
H4 {translate <train, Y4, 0> rotate <xr4,0,0> }
H5 {translate <train, Y5, 0> rotate <xr5,0,0> }
H6 {translate <train, Y6, O> rotate <xr6,0,0> }
H7 {translate <train, Y7, 0> rotate <xr7,0,0> }
H8 {translate <train, Y8, 0> rotate <xr8,0,0> }
M1 {translate <train, Y1, 0> rotate <-xr1,0,0> }
M2 {translate <train, Y2, 0> rotate <-xr2,0,0> }
M3 {translate <train, Y3, 0> rotate <-xr3,0,0> }
M4 {translate <train, Y4, 0> rotate <-xr4,0,0> }
M5 {translate <train, Y5, 0> rotate <-xr5,0,0> }
M6 {translate <train, Y6, 0> rotate <-xr6,0,0> }
M7 {translate <train, Y7, 0> rotate <-xr7,0,0> }
M8 {translate <train, Y8, 0> rotate <-xr8,0,0> }
M21 {translate <train, Y1, 0> rotate <xr1,0,0> }
M22 {translate <train, Y2, 0> rotate <xr2,0,0> }
M23 {translate <train, Y3, 0> rotate <xr3,0,0> }
M24 {translate <train, Y4, 0> rotate <xr4,0,0> }
M25 {translate <train, Y5, 0> rotate <xr5,0,0> }
M26 {translate <train, Y6, O> rotate <xr6,0,0> }
M27 {translate <train, Y7, 0> rotate <xr7,0,0> }
M28 {translate <train, Y8, 0> rotate <xr8,0,0> }
```

The left, middle, and right portions of each character are first grouped together to form single named coplanar objects (*H1-H8*, *M1-M8*, and *M21-M28*) that will move as units under rotations and translations. A copy of the first M is made and moved 131 units to the right to form the second one. As we mentioned earlier, the top-down order has been reversed to compensate for the way the letters were originally defined.

Two motions are imparted to these slices. First, each slice bounces up and down with simple harmonic motion every 15 frames (15*24=360°) using the variables Y1-Y8:

```
define major 0.2
define spacing 13

define Y1 0* spacing * major * sin(frame*24*rad)
define Y2 1* spacing * major * sin(frame*24*rad)
...
```

Next, the letters are made to rock front-to-back every 30 frames (30*6 = 360°) with varying amplitude, from 0° at the base for a stable platform to $\pm 17.5^{\circ}$ at the top (2.5*7 = 17.5°):

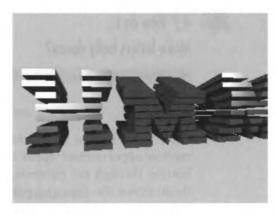


Figure 4-18 HMM train

```
define amp 2.5

define xr1 0 * amp * sin(rad * frame * 6)
define xr2 1 * amp * sin(rad * frame * 6)
...
define xr8 7 * amp * sin(rad * frame * 6)
```

Finally, the letters run past the camera at a rate of 4 units per frame using the variable *train*. For simplicity, we combine *train* with the up-and-down translation (*Y1-Y8*) in a single step:

```
define train 270-frame*4
...translate <train,Y1,0>...
```

A frame from the middle of the HMM.PI is shown in Figure 4-18.

Comments

Additional harmonics can be added to the motions of the letter elements to make them do odd phased accordion, stacking poker chip, polka silliness. For example, substituting the *Y1-Y8* variables with ones having an extra trig function, like

```
define major 0.2
define minor 0.05
define Y2 1*spacing*major*sin(frame*24*rad) + 1*minor*sin(frame*48*rad)
```

adds a little extra hop to every bounce. You can also precess each letter component like a top, and make them belly dance. There's enough extra detail in that one to cover it in the next section.



🚨 4.7 How do I...

Make letters belly dance?

You'll find the code for this in: PLY\CHAPTER4\WATUSI

Problem

Once we've broken up our letters into individual slices, a whole world of motion opportunities opens up for us. We can make a twist run top-to-bottom through the elements, or we can perform a phased precession on them. Given the time and patience, we might make them win the pommel horse competition at the 1996 Olympics. But for now, let's just make them shimmy.

Technique

We'll use the same HM.INC file created by HM.BAS in Section 4.6 for the data set. Then we'll spice up the image with colored spots, place it inside a sphere to capture the shadows easily, and rock the letter sections independently. This process is shown in the WATUSI.PI listing.

```
// WATUSI.PI - Hoochie-Koochie Letters
start_frame 0
end_frame 44
total_frames 45
outfile "wats"
define index 360*frame/total_frames
viewpoint {
   from <200,25,180>
   at <200,55,0>
  up <0,-1,0>
   angle 60
   resolution 320,120
   aspect 3
include "\PLY\COLORS.INC"
define shiny_white texture { shiny { color white } }
include "hm.inc"
define H8 object { HL1 + HR1 }
define H7 object { HL2 + HR2 }
define H6 object { HL3 + HR3 }
define H5 object { HL4 }
define H4 object { HL5 }
```

```
define H3 object { HL6 + HR6 }
define H2 object { HL7 + HR7 }
define H1 object { HL8 + HR8 }
define M18 object { ML1 + MR1 }
define M17 object { ML2 + MR2 }
define M16 object { ML3 + MR3 }
define M15 object { ML4 + MML4 + MMR4 + MR4 }
define M14 object { ML5 + MM5 + MR5 }
define M13 object { ML6 + MM6 + MR6 }
define M12 object { ML7 + MM7 + MR7 }
define M11 object { ML8 + MM8 + MR8 }
define M28 object { ML1 + MR1 translate <131,0,0>}
define M27 object { ML2 + MR2 translate <131,0,0>}
define M26 object { ML3 + MR3 translate <131,0,0>}
define M25 object { ML4 + MML4 + MMR4 + MR4 translate <131,0,0>}
define M24 object { ML5 + MM5 + MR5 translate <131,0,0>}
define M23 object { ML6 + MM6 + MR6 translate <131,0,0>}
define M22 object { ML7 + MM7 + MR7 translate <131,0,0>}
define M21 object { ML8 + MM8 + MR8 translate <131,0,0>}
define pi 3.14159
define rad pi/180
// center x locations for each letter
define HCX 70
define MC1 (189.21+210.79)/2
define MC2 MC1 + 131
define Y8 3.5
define Y7 16.5
define Y6 29.5
define Y5 42.5
define Y4 55.5
define Y3 68.5
define Y2 81.5
define Y1 94.5
// skewers for the letters - uncomment them if you'd like to see them
//object {cylinder <HCX,-50,0>,<HCX,150,0>,7 shiny_red }
//object {cylinder <MC1,-50,0>,<MC1,150,0>,7 shiny red }
//object {cylinder <MC2,-50,0>,<MC2,150,0>,7 shiny_red }
spot light coral *4, <HCX-200,Y4,200>,<HCX,Y4,0>,1,15,30
spot_light green *2, <MC1,Y4,200> ,<MC1,Y4,O>,1,15,40
spot_light blue *4, <MC2+200,Y4,200>,<MC2,Y4,O>,1,15,30
light red*0.5, <130, Y4, 0>
light red*0.5, <270, Y4, 0>
//light blue,<(HCX+MC2)/2,Y4,0>
```

continued on next page

```
continued from previous page
//light blue,<(MC1+MC2)/2,Y4,0>
define amp 7.5
// eight phased precessions
define xr1 amp * sin(rad * (index + 0*360/8))
define xr2 amp * sin(rad * (index + 1*360/8))
define xr3 amp * sin(rad * (index + 2*360/8))
define xr4 amp * sin(rad * (index + 3*360/8))
define xr5 amp * sin(rad * (index + 4*360/8))
define xr6 amp * sin(rad * (index + 5*360/8))
define xr7 amp * sin(rad * (index + 6*360/8))
define xr8 amp * sin(rad * (index + 7*360/8))
H1 {translate <-HCX,-Y1,0> rotate <xr3,0,xr1> translate <HCX,Y1,0> }
H2 {translate <-HCX,-Y2,0> rotate <xr4,0,xr2> translate <HCX,Y2,0> }
H3 {translate <-HCX,-Y3,0> rotate <xr5,0,xr3> translate <HCX,Y3,0> }
H4 {translate <-HCX,-Y4,0> rotate <xr6,0,xr4> translate <HCX,Y4,0> }
H5 {translate <-HCX,-Y5,0> rotate <xr7,0,xr5> translate <HCX,Y5,0> }
H6 {translate <-HCX,-Y6,0> rotate <xr8,0,xr6> translate <HCX,Y6,0> }
H7 {translate <-HCX,-Y7,0> rotate <xr1,0,xr7> translate <HCX,Y7,0> }
H8 {translate <-HCX,-Y8,0> rotate <xr2,0,xr8> translate <HCX,Y8,0> }
M11 {translate <-MC1,-Y1,0> rotate <xr6,0,xr4> translate <MC1,Y1,0> }
M12 {translate <-MC1,-Y2,0> rotate <xr7,0,xr5> translate <MC1,Y2,0> }
M13 {translate <-MC1,-Y3,0> rotate <xr8,0,xr6> translate <MC1,Y3,0> }
M14 {translate <-MC1,-Y4,0> rotate <xr1,0,xr7> translate <MC1,Y4,0> }
M15 {translate <-MC1,-Y5,0> rotate <xr2,0,xr8> translate <MC1,Y5,0> }
M16 {translate <-MC1,-Y6,0> rotate <xr3,0,xr1> translate <MC1,Y6,0> }
M17 {translate <-MC1,-Y7,0> rotate <xr4,0,xr2> translate <MC1,Y7,0> }
M18 {translate <-MC1,-Y8,0> rotate <xr5,0,xr3> translate <MC1,Y8,0> }
M21 {translate <-MC2,-Y1,0> rotate <xr1,0,xr7> translate <MC2,Y1,0> }
M22 {translate <-MC2,-Y2,0> rotate <xr2,0,xr8> translate <MC2,Y2,0> }
M23 {translate <-MC2,-Y3,0> rotate <xr3,0,xr1> translate <MC2,Y3,0> }
M24 {translate <-MC2,-Y4,0> rotate <xr4,0,xr2> translate <MC2,Y4,0> }
M25 (translate <-MC2,-Y5,0> rotate <xr5,0,xr3> translate <MC2,Y5,0> }
M26 {translate <-MC2,-Y6,0> rotate <xr6,0,xr4> translate <MC2,Y6,0> }
M27 {translate <-MC2,-Y7,0> rotate <xr7,0,xr5> translate <MC2,Y7,0> }
M28 (translate <-MC2,-Y8,0> rotate <xr8,5,xr6> translate <MC2,Y8,0> }
object { sphere <MC1,Y4,0>,2000 matte_white}
```

How It Works

Each letter element is assigned a *shiny_white* texture. Several brightly colored spotlights give us the colors we want. A large sphere around our model catches the shadows cast by letters in the intersecting spot lights (see Figure 4-19).

Eight phased terms are defined to precess the character elements, the magnitude being controlled by the scalar *amp*, set here to 7.5:

```
define amp 7.5

// eight phased precessions

define xr1 amp * sin(rad * (index + 0*360/8))
define xr2 amp * sin(rad * (index + 1*360/8))
define xr3 amp * sin(rad * (index + 2*360/8))
define xr4 amp * sin(rad * (index + 3*360/8))
```

Center Rotation

In order to rotate an element about its center, it must first be translated to the origin, rotated there, then translated back to its original position. Rotating objects not centered on the origin moves them around like leaves in a whirlwind. The code defining the letters originally calculated centers for each component. Getting the center of the entire letter from those pieces, however, proved more difficult than simply rendering guesses about their probable locations in the scene using red cylinders. They've been commented out in the following data file but feel free to render them if you're curious.

```
// center x locations for each letter.
define HCX 70
define MC1 (189.21+210.79)/2
define MC2 MC1 + 131

// skewers for the letters - uncomment them if you'd like to see them
//object {cylinder <HCX,-50,0>,<HCX,150,0>,7 shiny_red }
//object {cylinder <MC1,-50,0>,<MC1,150,0>,7 shiny_red }
//object {cylinder <MC2,-50,0>,<MC2,150,0>,7 shiny_red }
```

Since our letter slices are spaced so regularly, generating the height components Y1-Y8 for the y centers is easy. It is simply half the thickness of each letter (7/2) plus the spacing between the slices, or 3.5+n*13, where n is the slice number. Note that they too are specified in reverse order.

```
define Y8 3.5
define Y7 16.5
define Y6 29.5
define Y5 42.5
```



Figure 4-19 Spotlit letters doing an exotic dance

Each slice is called, translated to the origin, rotated, then translated back:

```
H1 {translate <-HCX,-Y1,0> rotate <xr3,0,xr1> translate <HCX,Y1,0> }
H2 {translate <-HCX,-Y2,0> rotate <xr4,0,xr2> translate <HCX,Y2,0> }
```

This animation precesses each element about the y axis. We can also twist each character around the y axis as we precess it for additional wiggly motion with the lines:

```
H1 {translate <-HCX,-Y1,0> rotate <xr3,xr5,xr1> translate <HCX,Y1,0> }
H2 {translate <-HCX,-Y2,0> rotate <xr4,xr6,xr2> translate <HCX,Y2,0> }
```



4.8 How do I...

Twist a louver ribbon around a bumping three-level blob?

You'll find the code for this in: PLY\CHAPTER4\THREAD

Problem

Ribbons that rotate, bend, and generally wander aimlessly about in 3-D space can perhaps be done using elaborate sets of triangle meshes, but it's a painful and tedious process. Take for example a flat coiled spring, thin along the dimension it would normally bounce up-and-down in (Figure 4-20).

Now let's rotate it about its long axis (see the arrow) in some continuous fashion. What would take massive brain scratching and vertex manipulation with a triangle mesh can be done much easier with a collection of Bezier

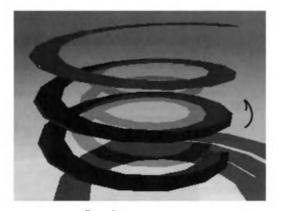


Figure 4-20 A flat coiled spring

patches. We'll describe those in more detail in a moment. The effect we're after is to use this rotating coil as a cylindrical venetian blind that periodically shuts off the light from a spotlight placed above it, showing up as a spiral shadow that opens and closes ("gills") on a plane placed below our coil. We'll also add a shiny bouncing blob as a reflective focal point, just to make it interesting.

Technique

Bezier patches are malleable geometric primitives that are created using 16 control points scattered in space that tug a surface towards some desired shape. They're vague and mysterious, but once you play with them for a while they're just another convenient way to define a surface. They make near warps rather than through warps, like the spline paths in Section 3.4. The points shepherd the surface toward the desired shape, but the control points aren't actually contained in the surface. An example of a Bezier patch with its control points is shown in Figure 4-21.

This animation places an S-shaped Bezier object rotating about itself in a spiral path around a bobbing blue reflective three-level blob. The S shape was an accident, caused by calling the control points in the wrong order. The desired shape was actually a spiraling car fender, or something like the threads on a light bulb, but the S looked fine, so it stayed. It has C2 symmetry, meaning it looks the same flipped 180°. An overhead light gets slatted on and off when the wider parts of the Bezier are perpendicular to the direction of the spotlight. While the entire animation only runs 45 frames, the reflections and the Aldus lamp type lighting make it really neat.



Figure 4-21 Bezier patch and its corresponding shepherd points represented by spheres

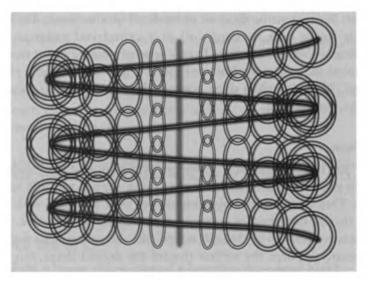


Figure 4-22 Bezier control points that rotate, surrounding a spiral path

Steps

The animation was originally done as a series of 45 include files created by QuickBasic, which also did the simulation code for the motion of the control points. As is usually the case with Polyray, it was possible to construct a single Polyray data file with the all the animation inside it, and dispense with a messy external animation routine. The following QuickBasic code illustrates the rotations of the control points. An illustration of the coiled Bezier control points that this program displays is shown in Figure 4-22. What's actually being shown are the orbits in which sets of four control points circulate. The points form circles because the screen wasn't cleared between each iteration.

```
'BEZIER.BAS

DECLARE SUB rotate (x, y, z)

COMMON SHARED rad, xrotate, yrotate, zrotate

TYPE Vector

x AS SINGLE

y AS SINGLE

z AS SINGLE

END TYPE

DIM p(4) AS Vector

pi = 3.1415927#

rad = pi / 180
```

```
SCREEN 12
WINDOW (-16, -4)-(16, 20)
CLS
' our four control points. These get spiraled
' down a coil to generate the Bezier Batches
p(1).x = 1: p(1).y = 1: p(1).z = 0
p(2).x = -1: p(2).y = 1: p(2).z = 0
p(3).x = 1: p(3).y = -1: p(3).z = 0
p(4).x = -1: p(4).y = -1: p(4).z = 0
' show a spiral path 2-D
   FOR a = 90 \text{ TO } 90 + 360 * 3
      y = 10 * SIN(a * rad)
      CIRCLE (y, (a / 75) - 1), .1, 1
      PAINT (y, (a / 75) - 1), 15, 1
   NEXT a
FOR dat = 0 TO 44 '45 frames
   xrotate = 0
   yrotate = 0
   zrotate = 4
   ' take 4 points in a square and rotate them
   FOR a = 1 TO 4
      CALL rotate(p(a).x, p(a).y, p(a).z)
   NEXT a
   r = 10
   ang = 0
                   ' initial value for ang = 0
   thread = 5 / 360 ' gives 5 units up per 360 degrees
   LOCATE 1, 1: PRINT "Frame ";dat
   FOR patch = 1 TO 24 ' 24 patches spanning 45 degrees each
      FOR n = 1 TO 4 ' 4 sets of 4 points per patch
         FOR a = 1 TO 4 ' each point
            xrotate = 0
            yrotate = ang ' rotate about y axis
            zrotate = 0
            tx = p(a).x + r
                                       ' offset sets a circle
            ty = p(a).y + ang * thread ' this gives a spiral
            tz = p(a).z + 0
            CALL rotate(tx, ty, tz)
            IF n = 2 OR n = 3 THEN
```

continued on next page

```
continued from previous page
                tx = tx * 1.04 ' move the middle two control point sets tz = tz * 1.04 ' out to make the spiral rounder
             END IF
             CIRCLE (tx, ty), .1, a + k
         NEXT a
          ' ang isn't indexed after the last set,
          ' so the Bezier ends will overlap
         IF n < 4 THEN ang = ang + (45 / 3)
      NEXT n
   NEXT patch
NEXT dat
SUB rotate (x, y, z)
'rotate
    x0 = x
    y0 = y
    z0 = z
    x1 = x0
    y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
    z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)
    x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
    y2 = y1
    z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)
    x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)
    y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)
    z3 = z2
    x = x3
    y = y3
    z = z3
```

Polyray Code

The Polyray code for the animation is broken into two parts. The first part sets the scene and generates the motion; the second defines the object being moved. The first part is easy.

```
// Spiraling Bezier Animation: Jeff Bowermaster
start_frame 0
end_frame 44
total_frames 45
```

END SUB

```
define index 360/total frames
outfile thred
include "\PLY\COLORS.INC"
// set up background color & lights
background midnightblue
light <0,25,0>
light <10,50,-120>
viewpoint {
   from <5.01, 40.01,-40.01>
   at <0,0,7.5>
        <0, 1, 0>
   up
   angle 30
   resolution 320,200
   aspect 1.433
   }
// make a tabletop
define table
object {
   polygon 4,<-25,0.01,-25>,<25,0.01,-25>,<25,0.01,25>,<-25,0.01,25>
   reflective_blue
)
object {
   box <-25,-1,-25>,<25,0,25>
   texture {surface { color navyblue ambient 0.2 diffuse 0.6}}
)
define pi 3.14159
define rad pi/180
// blobs 3-phased height components - each 1/3 out from the other
define a 7 + 3*sin(index*frame*rad)
define b 7 + 3*sin(index*(frame+(total_frames/3)*rad))
define c 7 + 3*sin(index*(frame+(2*total_frames/3)*rad))
object {
   blob 0.6:
      sphere <0,a,0>,3,3,
      sphere <0,b,0>, 4, 4,
      sphere <0,c,0>,5,5
   reflective_cyan
}
// rotate the base Bezier patch
define p1 rotate(<1,1,0>,<0,0,frame*index/2>)
define p2 rotate(<-1,1,0>,<0,0,frame*index/2>)
define p3 rotate(<1,-1,0>,<0,0,frame*index/2>)
define p4 rotate(<-1,-1,0>,<0,0,frame*index/2>)
                                                                  continued on next page
```

The file THREAD.INC in the last line contains 24 sets of 16-point control vertices (384 points total) which will make up our spiraling ribbon. It would be insane to try to write all this information out manually. The file THREAD.BAS lets QuickBasic do it for us, using shortcuts based on regular features of our object:

```
' THREAD.BAS - Twisting Ribbon Creation code
OPEN "thread.inc" FOR OUTPUT AS #1
FOR p1 = 0 TO 23
FOR p2 = 0 TO 3
   var$ = RIGHT$("00" + LTRIM$(STR$(p1)), 2) + RIGHT$("00" + LTRIM$(STR$(p2)), 2)
   PRINT #1, USING "define pch\\ rotate(p# + <r, ang*thread, 0>,<0, ang, ←
0>)"; var$; p2 MOD 4 + 1
NEXT p2
PRINT #1,
PRINT #1, "static define ang ang+15"
PRINT #1,
FOR p2 = 4 TO 7
   var$ = RIGHT$("00" + LTRIM$(STR$(p1)), 2) + RIGHT$("00" + LTRIM$(STR$(p2)), 2)
   PRINT #1, USING "static define pch\\ rotate(p# + <r, ang*thread, 0>,<0, \Leftarrow
ang, 0 > )"; var$; p2 MOD 4 + 1
   PRINT #1, USING "define pcha\ \ [pch\ \]"; var$; var$
   PRINT #1, USING "define addl\ \ <pcha\ \[0][0]*offs,0,pcha\ \Leftarrow
\[0][2]*offs>"; var$; var$; var$
   PRINT #1, USING "static define pch\ \ pch\ \+addl\ \"; var$; var$; var$
NEXT p2
PRINT #1,
PRINT #1, "static define ang ang+15"
PRINT #1,
FOR p2 = 8 TO 11
   var$ = RIGHT$("00" + LTRIM$(STR$(p1)), 2) + RIGHT$("00" + LTRIM$(STR$(p2)), 2)
   PRINT #1, USING "static define pch\ \ rotate(p# + <r, ang*thread, 0>,<0, \Leftarrow
ang, 0>)"; var$; p2 MOD 4 + 1
   PRINT #1, USING "define pcha\ \ [pch\ \]"; var$; var$
   PRINT #1, USING "define addl\ \ <pcha\ \[0][0]*offs,0,pcha\ \equiv \]
\[0][2]*offs>"; var$; var$; var$
   PRINT #1, USING "static define pch\ \ pch\ \+addl\ \"; var$; var$; var$
```

```
NEXT p2
PRINT #1,
PRINT #1, "static define ang ang+15"
PRINT #1,
FOR p2 = 12 \text{ TO } 15
   var$ = RIGHT$("00" + LTRIM$(STR$(p1)), 2) + RIGHT$("00" + LTRIM$(STR$(p2)), 2)
   PRINT #1, USING "define pch\\ rotate(p# + <r, ang*thread, 0>,<0, ang, \Leftarrow
0>)"; var$; p2 MOD 4 + 1
NEXT p2
NEXT p1
FOR p1 = 0 TO 23
PRINT #1, "object {"
PRINT #1, " bezier 2, detail, 10, 10,"
   FOR p2 = 0 TO 15
      var$ = "pch" + RIGHT$("00" + LTRIM$(STR$(p1)), 2) + RIGHT$("00" + \Leftarrow 
LTRIM\$(STR\$(p2)), 2)
      PRINT #1, var$;
      IF p2 < 15 THEN PRINT #1, ",";
   NEXT p2
   PRINT #1, " reflective_coral"
   PRINT #1, "}";
   PRINT #1,
NEXT p1
CLOSE #1
           The first of the 24 patches that THREAD.BAS generates follows:
define pch0000 rotate(p1 + <r, ang*thread, 0>,<0, ang, 0>)
define pch0001 rotate(p2 + <r, ang*thread, 0>,<0, ang, 0>)
define pch0002 rotate(p3 + <r, ang*thread, 0>,<0, ang, 0>)
define pch0003 rotate(p4 + <r, ang*thread, 0>,<0, ang, 0>)
static define ang ang+15
define pch0004 rotate(p1 + <r, ang*thread, 0>,<0, ang, 0>)
static define pcha0004 [pch0004]
define addl0004 <pcha0004[0][0]*offs,0,pcha0004[0][2]*offs>
static define pch0004 pch0004+addl0004
define pch0005 rotate(p2 + \langle r, ang*thread, 0 \rangle, \langle 0, ang, 0 \rangle)
static define pcha0005 [pch0005]
define addl0005 <pcha0005[0][0]*offs,0,pcha0005[0][2]*offs>
static define pch0005 pch0005+addl0005
define pch0006 rotate(p3 + <r, ang*thread, 0>,<0, ang, 0>)
static define pcha0006 [pch0006]
define addl0006 <pcha0006[0][0]*offs,0,pcha0006[0][2]*offs>
static define pch0006 pch0006+addl0006
define pch0007 rotate(p4 + <r, ang*thread, 0>,<0, ang, 0>)
                                                                    continued on next page
```

```
continued from previous page
static define pcha0007 [pch0007]
define addl0007 <pcha0007[0][0]*offs,0,pcha0007[0][2]*offs>
static define pch0007 pch0007+addl0007
static define ang ang+15
define pch0008 rotate(p1 + <r, ang*thread, 0>,<0, ang, 0>)
static define pcha0008 [pch0008]
define addl0008 <pcha0008[0][0]*offs,0,pcha0008[0][2]*offs>
static define pch0008 pch0008+addl0008
define pch0009 rotate(p2 + <r, ang*thread, 0>,<0, ang, 0>)
static define pcha0009 [pch0009]
define addl0009 <pcha0009[0][0]*offs,0,pcha0009[0][2]*offs>
static define pch0009 pch0009+addl0009
define pch0010 rotate(p3 + <r, ang*thread, 0>,<0, ang, 0>)
static define pcha0010 [pch0010]
define addl0010 <pcha0010[0][0]*offs,0,pcha0010[0][2]*offs>
static define pch0010 pch0010+addl0010
define pch0011 rotate(p4 + <r, ang*thread, 0>,<0, ang, 0>)
static define pcha0011 [pch0011]
define addl0011 <pcha0011[0][0]*offs,0,pcha0011[0][2]*offs>
static define pch0011 pch0011+addl0011
static define ang ang+15
define pch0012 rotate(p1 + <r, ang*thread, 0>,<0, ang, 0>)
define pch0013 rotate(p2 + <r, ang*thread, 0>,<0, ang, 0>)
define pch0014 rotate(p3 + <r, ang*thread, 0>,<0, ang, 0>)
define pch0015 rotate(p4 + <r, ang*thread, 0>,<0, ang, 0>)
```

How It Works

This code really only has two jobs to do. The first thing it does is generate some valid Polyray string variables, complete with the appropriate rotation and translation syntax, to place the patch control points around our spiraling path. This is handled with code like

The numbers specifying the patch number and point number are generated using a number-to-text conversion of the counting variables p1 and p2, and placed where the "\ " occurs in the line:

```
FOR p1 = 0 TO 23

FOR p2 = 0 TO 3

var$ = RIGHT$("00" + LTRIM$(STR$(p1)), 2)

+ RIGHT$("00" + LTRIM$(STR$(p2)), 2)
```

The actual point being rotated is placed where the # is (rotate (p# ...)).

These points are displaced out from a center line by r, moved down a path by ang*thread, then rotated about an angle we index 15° for each set of four. Since Bezier patches have to meet at their ends to make solid surfaces, the angle is not indexed after the last set, meaning the first four points for the next patch will overlap these. It just keeps going as we move down the thread. Nothing special here at all.

The second thing it does gets nasty and involved, but it's necessary because without it the shiny surface of our ribbon will reveal joints where the Bezier patches meet, and we don't want that. In order for the curvature of the Beziers to be consistent at their junctions, the two middle sets of four points must be shifted out about 4% from a line running down the center of our spiral, and that's where the code gets nasty. We need to shift the x and z values of the vector for each point, but leave their y value alone. This requires converting vectors to scalars.

Vectors and Scalars

There are two types of variables in Polyray: vectors and scalars. Vectors are collections of three numbers, usually either the x, y, and z coordinates or the RGB red-green-blue values for a color. Scalars are single numbers. What happens when you need to multiply only part of a vector by a scalar? You convert the vector to a scalar and deal with the values individually. Consider the next four lines:

```
static define pch0004 rotate(p1 + <r, ang*thread, 0>,<0, ang, 0>) define pcha0004 [pch0004] define addl0004 <pcha0004[0][0]*offs,0,pcha0004[0][2]*offs> static define pch0004 pch0004+addl0004
```

We first define the vector pch0004 as static, to allow operations equivalent to QuickBasic's x = x + 1. Mathematically this statement can never be true, but it is the standard incrementing syntax for Basic. Next, the vector pch0004 is copied into a format that grabs individual pieces of it. Once set as

```
define pcha0004 [pch0004]
```

the individual components of the vector can be defined as pcha0004E0JE0J, pcha0004E0JE1J, and pcha0004E0JE2J

We multiply the x and z portions by offs, make a vector called addl0004 with the offset required to round out our ring, then add this vector to the original point pch0004. It may seem like a long way to go to get a small effect, but try running the animation with offs set to 0.0 and 0.04 and see what you think.



Figure 4-23 Blob ringed by twisting Bezier patches

We treat the second and third set of four Bezier points differently from the first and last set by having four separate loops inside the main one. The variable *ang* controll our location on the spiral. It is stepped 15° for each patch (except the last one) with static define ang

static define ang ang+15

making the end points of each patch overlap, as seen in Figure 4-23.

Comments

This code may at first seem to be a long way to go to generate a fairly simple effect. The advantage of this method, however, is that by nesting additional control code, it is possible to add thousands of coils with all sorts of bends and twists and scaling that would be impractical if done manually. This way more or less herds the animation in the direction you want it to go.



4.8 How do I...

Make a rubbery hyperactive cube?

You'll find the code for this in: PLY\CHAPTER4\BOX

Problem

Making collections of objects connected by springs move realistically presents a real challenge. It is usually done with a process known as finite element analysis (FEA). FEA considers each object's momentum, the forces it exerts on the other objects in the ensemble, and a stiffness matrix to generate

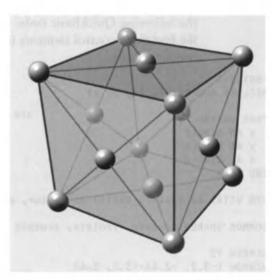


Figure 4-24 Rubbery cube with spheres at the eight corners and six face centers, all tied together

appropriate motions from the applied forces. Fortunately, phased motions can generate a reasonable facsimile without resorting to FEA.

Technique

Take spheres representing both the corners and face centers of a cube (Figure 4-24), connect them with rods, then make all these spheres breathe in and out following a simple sine wave motion. Rather than making all the corners or face centers move at the same time, however, the motions are phased so that each point moves at different times. Due to the definition sequence (corners first, face centers last), the corners move out of phase with the face centers, so the figure does this disjointed bouncy dance. It looks like a rag doll in the hands of a tantrum throwing expert.

While the smart way to generate a decent loop from this animation involves deep thought and probably several massively parallel computers, a simpler way is to generate a few hundred frames and pick over them until a reasonable loop point is found. Frames 236 and 710 seemed to work particularly nicely.

QuickBasic Simulation

In order to determine appropriate values for the amplitude of the oscillations and the way we'd like to see the figure tumble, we simulate our motion with

the following QuickBasic code. It has been abbreviated here to include only the first three control elements in each section. See the BOX.BAS file for the complete listing.

```
'BOX.BAS
DECLARE SUB Rotate (x, y, z)
TYPE Vector
 x AS SINGLE
 y AS SINGLE
 z AS SINGLE
END TYPE
DIM V(14) AS Vector, Vo(14) AS Vector, phase(14), red(16), green(16), blue(16)
COMMON SHARED xrotate, yrotate, zrotate
SCREEN 12
WINDOW (-3.2, -2.4)-(3.2, 2.4)
FOR y = 1 TO 4
       FOR x = 1 TO 4
                colornum = x + ((y - 1) * 4) - 1
               READ red(colornum), green(colornum), blue(colornum)
               KOLOR = 65536 * blue(colornum) + 256 * green(colornum) + red(colornum)
               PALETTE colornum, KOLOR
               COLOR colornum
       NEXT x
NEXT y
'rainbow palette
DATA 16, 0, 0
DATA 32, 0, 0
DATA 42, 0, 0
DATA 58, 16, 0
DATA 63, 32, 0
DATA 58, 56, 0
DATA 16, 42, 0
DATA 0, 30, 36
DATA 0, 20, 40
DATA 0, 10, 48
DATA 0, 0, 63
DATA 20, 0, 53
DATA 23, 0, 29
DATA 19, 7, 17
DATA 50, 40, 45
DATA 63, 63, 63
startframe = 236
```

```
endframe = 710
stp = .75
f = 1
ampl = .15
DO WHILE INKEY$ = ""
FOR frame = startframe TO endframe STEP stp
pi = 3.14159
rad = pi / 180
rot = frame * stp
phase01 = 360 * 1 / 14
phase02 = 360 * 2 / 14
phase03 = 360 * 3 / 14
a01 = SIN((rot + phase01) * f) * stp * ampl
a02 = SIN((rot + phase02) * f) * stp * ampl
a03 = SIN((rot + phase03) * f) * stp * ampl
v01.x = -1 + a01: v01.y = -1 + a01: v01.z = -1 + a01
v02.x = 1 + a02: v02.y = -1 + a02: v02.z = -1 + a02
v03.x = 1 + a03: v03.y = 1 + a03: v03.z = -1 + a03
xrotate = 155 * SIN(rot * rad)
yrotate = 240 * COS(rot * rad)
zrotate = 190 * SIN(rot * rad)
CALL Rotate(v01.x, v01.y, v01.z)
CALL Rotate(v02.x, v02.y, v02.z)
CALL Rotate(v03.x, v03.y, v03.z)
'undraw
LINE (vo01.x, vo01.y)-(vo02.x, vo02.y), 0
LINE (vo02.x, vo02.y)-(vo09.x, vo09.y), 0
LINE (vo09.x, vo09.y)-(vo01.x, vo01.y), 0
. . .
'draw
LINE (v01.x, v01.y)-(v02.x, v02.y), 1
LINE (v02.x, v02.y)-(v09.x, v09.y), 1
LINE (v09.x, v09.y)-(v01.x, v01.y), 1
vo01.x = v01.x
```

continued on next page

CHAPTER FOUR

```
continued from previous page
vo02.x = v02.x
vo03.x = v03.x
. . .
vo01.y = v01.y
vo02.y = v02.y
vo03.y = v03.y
. . .
NEXT frame
L00P
SUB Rotate (x, y, z)
       pi = 3.14159
       rad = pi / 180
       x0 = x
       y0 = y
       z0 = z
       x1 = x0
       y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
       z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)
       x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
       y2 = y1
       z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)
       x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)
       y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)
       z3 = z2
       x = x3
       y = y3
       z = z3
```

END SUB

244

How It Works

Fourteen equidistant phase angles, spaced between 0 and 360°, are generated for our 14 objects, making the eight corners and six face centers of the cube. These offset 14 sine waves will control the breathing motions of the vertices:

```
phase01 = 360 * 1 / 14
phase02 = 360 * 2 / 14
phase03 = 360 * 3 / 14
   ...

a01 = SIN((rot + phase01) * f) * stp * ampl
a02 = SIN((rot + phase02) * f) * stp * ampl
a03 = SIN((rot + phase03) * f) * stp * ampl
   ...
```

The tips and face centers of the cube are defined by specifying their static positions, and the phased components (a01-a14) are then added to each of these:

```
v01.x = -1 + a01: v01.y = -1 + a01: v01.z = -1 + a01

v02.x = 1 + a02: v02.y = -1 + a02: v02.z = -1 + a02

v03.x = 1 + a03: v03.y = 1 + a03: v03.z = -1 + a03

...
```

Since the same value is added to the x, y, and z components of each point, we create a simple in-and-out breathing motion. The phasing makes this motion occurs at different times for each vertex in our cube. The remainder of the code draws and undraws the connections between the points in our figure. The Polyray code for this animation (BOX.FLI) is scripted as follows.

```
// box:
          A Box Squirms and Tumbles
11
start_frame 236
end_frame 710
total_frames 486
outfile "box"
// set up the camera
viewpoint {
   from <0,2,8>
   at <0,0,0>
   up <0,1,0>
   angle 25
   resolution 160,120
   aspect 1.33
// get various surface finishes
include "\PLY\COLORS.INC"
// Set up background color & lights
background SkyBlue
// a spotlight
spot_light <3,3,3>, <0,10,5>, <0,5,0>, 4, 6, 12
// another light
light white, <10, 10, 10>
define stp
              0.75
define f
              1.0
define ampl
              0.15
define pi
              3.14159
define rad
              pi / 180
define rot frame*stp
```

CHAPTER FOUR

```
continued from previous page
define phase01 360 * 1 / 14
define phase02 360 * 2 / 14
define phase03 360 * 3 / 14
define a01 sin((rot + phase01) * f) * stp * ampl
define a02 sin((rot + phase02) * f) * stp * ampl
define a03 sin((rot + phase03) * f) * stp * ampl
// points
define v01 < -1+a01, -1+a01, -1+a01 >
define v02 < 1+a02, -1+a02, -1+a02 >
define v03 < 1+a03, 1+a03, -1+a03 >
define v04 < -1+a04, 1+a04, -1+a04 >
define v05 < -1+a05, -1+a05, 1+a05 >
define v06 < 1+a06, -1+a06, 1+a06 >
define v07 < 1+a07, 1+a07, 1+a07 >
define v08 < -1+a08, 1+a08, 1+a08 >
//centers
define v09 < 0+a09, 0+a09, -1+a09 >
define v10 < 0+a10, 0+a10, 1+a10 >
define v11 < 0+a11, -1+a11, 0+a11 >
define v12 < 0+a12, 1+a12, 0+a12 >
define v13 < -1+a13, 0+a13, 0+a13 >
define v14 < 1+a14, 0+a14, 0+a14 >
// single orbit, superimpose a modulated sinewave of a higher frequency
// wave that circles the sphere
define xrotate 155 * sin(rot*rad)
define yrotate 240 * cos(rot*rad)
define zrotate 190 * sin(rot*rad)
// our stick figure cube
object {
   object { cylinder v02, v09, 0.05 shiny_red } +
   object { cylinder v03, v09, 0.05 shiny_red } +
   object { cylinder v04, v09, 0.05 shiny_red } +
   object { cylinder v01, v09, 0.05 shiny_red } +
   object { sphere v01, 0.2 shiny_coral } +
   object { sphere v02, 0.2 shiny_coral } +
   object { sphere v03, 0.2 shiny_coral } +
  rotate rotate,yrotate,zrotate >
}
```

BOX.FLI is virtually identical to the QuickBasic simulation code, and works in exactly the same way. All the objects are collected into a single composite object and tumble the entire mass using a rotate command, as shown here:

```
define xrotate 155 * sin(rot*rad)
define yrotate 240 * cos(rot*rad)
define zrotate 190 * sin(rot*rad)
...
rotate <xrotate,yrotate,zrotate >
```

A sample frame from the animation is shown in Figure 4-25.

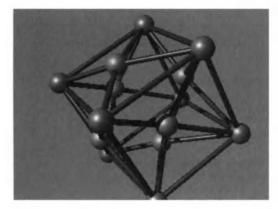


Figure 4-25 Stick-figure bouncing box

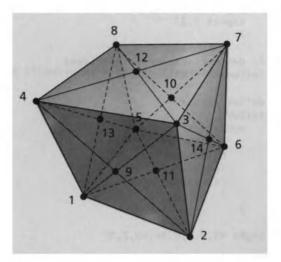


Figure 4-26 Cube numbered to facilitate triangle mesh definition

Modifications

We can use this same code to create a solid box rather than a stick figure, and just for fun we can use vector notation to define a series of triangle patches. There are 14 vertices that control the positions of 24 triangular patches (four per side, six sides). We can hide the vagueness of the vertex numbering inside vector notation for slightly neater code.

Organizing the vertex numbering and calling order requires that you draw the box, number the vertices, and keep track of where you are (Figure 4-26). It can be a lengthy process, but it's fairly straightforward, as shown in the next example Polyray file BOX2.FLI. Try to follow how each side is called based on the numbers shown in Figure 4-26 and the triangle definitions *t01-t24*.

```
// box2: A Mirror Ball Box Animation
11
            Jeff Bowermaster
start_frame 236
end_frame 710
total_frames 720
outfile "box2"
// Set up the camera
viewpoint {
   from <0,2,8>
   at <0,0,0>
   up <0,1,0>
   angle 25
   resolution 160,120
   aspect 1.33
// get various surface finishes
include "\PLY\COLORS.INC"
define chrome
texture {
   surface {
      ambient white, O.
      diffuse white, 0.1
      specular 1
      reflection white, 1
   }
light <1,0.5,0.3>,<0,2,8>
define stp
              0.75
define f
              1.0
define ampl
              0.25
define pi
              3.14159
```

```
define rad pi / 180
define rot frame*stp
define phase01 360 * 1 / 14
define phase02 360 * 2 / 14
define phase03 360 * 3 / 14
define a01 sin((rot + phase01) * f) * stp * ampl
define a02 sin((rot + phase02) * f) * stp * ampl
define a03 sin((rot + phase03) * f) * stp * ampl
// points
define v01 < -1+a01, -1+a01, -1+a01 >
define v02 < 1+a02, -1+a02, -1+a02 >
define v03 < 1+a03, 1+a03, -1+a03 >
define v04 < -1+a04, 1+a04, -1+a04 >
define v05 < -1+a05, -1+a05, 1+a05 >
define v06 < 1+a06, -1+a06, 1+a06 > define v07 < 1+a07, 1+a07, 1+a07 >
define v08 < -1+a08, 1+a08, 1+a08 >
//centers
define v09 < 0+a09, 0+a09, -1+a09 >
define v10 < 0+a10, 0+a10, 1+a10 >
define v11 < 0+a11, -1+a11, 0+a11 >
define v12 < 0+a12, 1+a12, 0+a12 >
define v13 < -1+a13, 0+a13, 0+a13 >
define v14 < 1+a14, 0+a14, 0+a14 >
// single orbit, superimpose a modulated sinewave of a higher frequency
// wave that circles the sphere
define xrotate 155 * sin(rot*rad)
define yrotate 240 * cos(rot*rad)
define zrotate 190 * sin(rot*rad)
// front
define t01 [v01,v02,v09]
define t02 [v02,v03,v09]
define t03 [v03,v04,v09]
define t04 [v04,v01,v09]
// back
define t05 [v06,v05,v10]
define t06 [v07,v06,v10]
define t07 [v08,v07,v10]
define t08 [v05,v08,v10]
                                                                  continued on next page
```

CHAPTER FOUR

```
continued from previous page
// bottom
define t09 [v05,v06,v11]
define t10 [v06,v02,v11]
define t11 [v02,v01,v11]
define t12 [v01,v05,v11]
// top
define t13 [v07,v08,v12]
define t14 [v08,v04,v12]
define t15 [v04,v03,v12]
define t16 [v03,v07,v12]
// left
define t17 [v08,v05,v13]
define t18 [v05,v01,v13]
define t19 [v01,v04,v13]
define t20 [v04,v08,v13]
// right
define t21 [v06,v07,v14]
define t22 [v07,v03,v14]
define t23 [v03,v02,v14]
define t24 [v02,v06,v14]
object {
   object { polygon 3, t01[0], t01[1], t01[2] } +
   object { polygon 3, t02[0], t02[1], t02[2] } +
   object { polygon 3, t03[0], t03[1], t03[2] } +
   object { polygon 3, t04[0], t04[1], t04[2] } +
   object { polygon 3, t05[0], t05[1], t05[2] } +
   object { polygon 3, t06[0], t06[1], t06[2] } +
   object { polygon 3, t07[0], t07[1], t07[2] } +
   object { polygon 3, t08[0], t08[1], t08[2] } +
   object { polygon 3, t09[0], t09[1], t09[2] } +
   object { polygon 3, t10[0], t10[1], t10[2] } +
   object { polygon 3, t11[0], t11[1], t11[2] } +
   object { polygon 3, t12[0], t12[1], t12[2] } +
   object { polygon 3, t13[0], t13[1], t13[2] } +
   object { polygon 3, t14[0], t14[1], t14[2] } +
   object { polygon 3, t15[0], t15[1], t15[2] } +
   object { polygon 3, t16[0], t16[1], t16[2] } +
   object { polygon 3, t17[0], t17[1], t17[2] } +
   object { polygon 3, t18[0], t18[1], t18[2] } +
   object { polygon 3, t19[0], t19[1], t19[2] } +
   object { polygon 3, t20[0], t20[1], t20[2] } +
   object { polygon 3, t21[0], t21[1], t21[2] } +
   object { polygon 3, t22[0], t22[1], t22[2] } +
   object { polygon 3, t23[0], t23[1], t23[2] } +
   object { polygon 3, t24[0], t24[1], t24[2] }
   texture {
      checker matte_magenta, matte_orange
      translate <0, -0.1, 0>
      scale <0.2, 0.2, 0.2>
```

```
}
rotate <xrotate, yrotate, zrotate>
}
object {
    sphere <0, 0, 0>, 9
    chrome
    }
object {
    sphere <0, 0, 0>, 1.2
    chrome
}
```

The object is defined as 24 triangular patches in vector notation. It's actually just a way of keeping point locations tidy. For example, the first triangle is defined as three points:

```
define t01 [v01,v02,v09]
```

This definition is subsequently used to define the polygon using the matrix notation:

```
object { polygon 3, t01[0], t01[1], t01[2] }
```

The substitution here is v01 = t01[0], v02 = t01[1], and v09 = t01[2], but it could just as easily have been done with

```
object { polygon 3, v01, v02, v09 }
```

The difference is cosmetic. We've spruced up the model a bit by placing the object between two concentric mirrored spheres. The inner one shows how we're moving the cube back and forth, and the outer one reflects the scene back to the inner one, producing the illusion that the cube is hollow. The motion is however the same as our stick figure cube. A sample frame is shown in Figure 4-27.

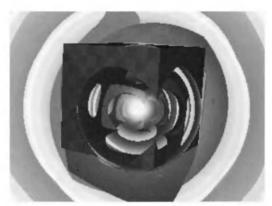
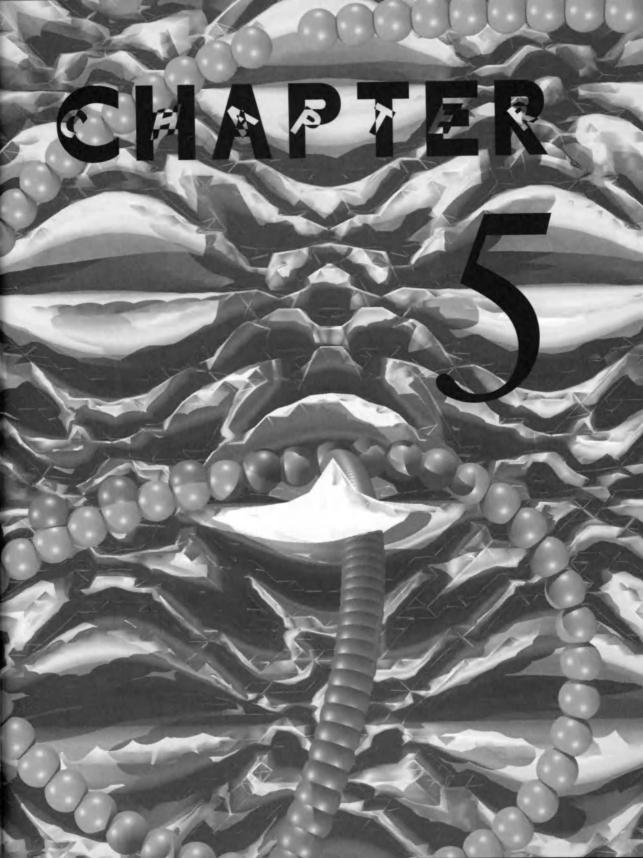


Figure 4-27 Box in between concentric mirrored spheres



5

PARTICLE SYSTEMS

article system animations are different from the kind of systems we've dealt with so far, since rather than directly controlling the shapes or motions of objects in a scene, you define rules which govern how objects move and the shapes they assume. In a sense, it's very similar to the classic cellular automata program LIFE. In LIFE, you define rules for cell propagation and some initial conditions and then sit back and watch little pixel colonies grow and retreat, evolve and develop, controlled by the rules you selected.

Particle systems that simulate things like fountains, fireworks, and gas dynamics follow very simple rules of motion. They involve incrementing the position of an object based on its velocity, and incrementing its velocity by some acceleration that may be controlled by the object's distance from some fixed point or by forces it might experience that emanate from other objects in the scene. Arrays hold the status of variables such as position, velocity and momentum for each object. At every clock tick the objects are moved, checked for interactions, and the controlling variables are modified accordingly.

Unpredictability

As these animations follow the natural laws of motion, particle systems give a wonderful sense of reality that objects forced to move along predetermined spline paths lack. Particle systems show the unpredictability of nature itself. The downside to this realism and unpredictability is that smooth loop points are difficult to locate, since the future position, direction or shape of an object depends on all the little intervening steps. Free-running animations created in real time don't need to loop, since they just continue on forever. However, all ray traced animations take much longer to render than to play, and discontinuous loop points can cause breaks in the motion.

Finding loop points with particle systems animation is usually a matter of trial and error and a good deal of luck. You can set up lengthy Monte Carlo simulations, where you randomly pick initial conditions, specify the loop criteria (i.e., the objects must all be within a certain distance from their starting points and be moving in roughly the same direction to qualify as a loop point), then run your simulation thousands and thousands of times until you stumble across an acceptable solution by accident. All you need to do is save your initial conditions for all these cases so that you can re-create the ones that worked.



🏅 5.1 How do I...

Animate objects with a free form collision based model?

You'll find the code for this in: PLY\CHAPTER5\BUMPERS

Problem

Free-running animations start with some initial conditions and generate motion based on conditions that arise as they run. One condition of practical

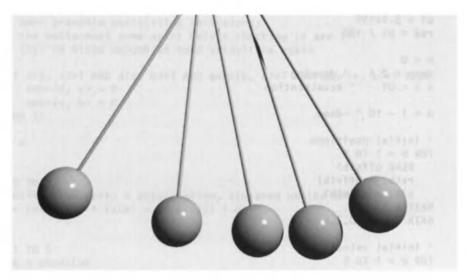


Figure 5-1 Random Bolita

interest is collisions. When the distance between any two objects falls below some minimum, code is envoked that handles this condition and takes appropriate action, such as exchanging the velocities and directions of the objects involved.

Technique

The following simulation (BUMPERS.BAS) doesn't contain an associated animation, although with external data files and batch processing we could generate one. Our purpose here is to demonstrate free-form collisions, and as a wire frame depiction, it can be given initial conditions and runs immediately. If Polyray had loops, this would be a much simpler animation to create. Bolita in Section 2.3 was a simple animation controlled by predictable functions. We can use the same model, except this time leave enough room in between the balls for randomness to set in (see Figure 5-1). Rather than forcing it to behave with sinusoidal motion, allow it to free-run. We need to determine when collisions between balls occur, and transfer momentum between them when they do.

```
' BUMPERS.BAS
' Five separated pendulums interact with free-form collisions

SCREEN 12
WINDOW (-12, -4)-(12, 14)
DIM embr(5, 5), d(5, 5)
```

continued on next page

CHAPTER FIVE

```
continued from previous page
   pi = 3.14159
   rad = pi / 180
   n = 0
   damp = 2.4 ' damping
   a = -.01 'acceleration
   d = 1 - 10 ^ -damp
   ' initial positions
   FOR b = 1 TO 5
      READ offx(b)
      rx(b) = offx(b)
      x(b) = offx(b)
   NEXT b
   DATA -4,-2, 0,2,4
   ' initial velocities
   FOR b = 1 TO 5
      READ v(b)
      rv(b) = v(b)
   NEXT b
   DATA .0,.0,.0,.0,.5
   ' embrace conditional setup
   FOR b = 1 TO 5
      FOR c = 1 TO 5
         embr(b, c) = 0
      NEXT c
   NEXT b
DO WHILE INKEY$ = ""
   ' move all the objects
   FOR b = 1 TO 5
      x(b) = x(b) + v(b)
   NEXT b
   FOR b = 1 TO 5
      FOR c = 1 TO 5
          ' distance between any two balls
          d(b, c) = ((x(b) - x(c)) ^ 2) ^ .5
          ' if two balls hit, swap their velocities
          IF d(b, c)<1 AND d(b, c)>0 AND embr(b, c)=0 AND embr(c, d)=0 THEN
                 temp = v(b)
                 v(b) = v(c)
                 v(c) = temp
                 embr(b, c) = 1
                 embr(c, b) = 1
          END IF
```

```
' embr prevents oscillating collisions;
          ' the balls must come apart before checking to see if
          ' they're close enough to swap velocities again
          IF d(b, c)>1 AND d(c, b)>1 AND embr(b, c)=1 AND embr(c, b)=1 THEN
             embr(b, c) = 0
             embr(c, b) = 0
          END IF
       NEXT c
   NEXT b
   FOR b = 1 TO 5
      ' velocity = velocity + acceleration, centered on the offset
      v(b) = (v(b) + a * (x(b) - offx(b))) * d
   NEXT b
   FOR b = 1 TO 5
      ' fake a pendulum
      cy(b) = .07 * (rx(b) - x(b)) ^ 2
      LINE (rx(b), 8)-(ox(b), ocy(b)), 0 ' undraw the current screen
      CIRCLE (ox(b), ocy(b)), .5, 0
                                          ' (draw in color 0 = black)
      LINE (rx(b), 8)-(x(b), cy(b)), 15
                                          ' draw the current screen
      CIRCLE (x(b), cy(b)), .5, 15
                                          ' (draw in color 15 = white)
      ox(b) = x(b)
                                           ' save the current screen
      oy(b) = Y(b)
                                          ' variables (ox=old-x; oy=old-y)
      ocy(b) = cy(b)
  NEXT b
L00P
```

How It Works

We establish the initial positions x(b) and velocities v(b) for our pendulums, specify values for acceleration and damping, and set our animation into motion by incrementing each object's location by its velocity. During each pass, a collision is looked for and acted upon on the basis of two factors:

- Are two spheres close enough to have collided?
- Is this the first time this condition has been true recently?

Once a collision has been detected, we swap the velocities of the objects. A problem with this approach is that if their rebound doesn't completely separate them by the next pass, we'll turn right around and swap their velocities again, resulting in an oscillatory embrace. We prevent this by using the variable *embr* as a toggle. Once two objects collide, they must come completely apart for at least one cycle before we check for another collision.

We shift the heights of the balls as they rock back and forth away from their rest positions rx(b) using a standard equation for a circle.

Comments

You might try changing the initial conditions, the acceleration, the damping, or the balls' initial velocities, and watch the results. Going to two dimensions (x and y) and doing a collision detection on walls as well as on the balls produces a very nice pool table simulation.



🏂 5.2 How do I...

Play "red light-green light" with a bunch of spheres?

You'll find the code for this in: PLY\CHAPTER5\REDLIGHT

Problem

Producing motion that's discontinuous but roughly coordinated for a collection of objects in motion would be difficult to manage using functions. Particle systems make it easy to accomplish this task.

Technique

The classic game of "red light-green light," played by several hundred million children, a Magician in a Bugs Bunny cartoon, and at least one very loud punk rock band, involves motion, no motion, motion, no motion... based on whether the control variable is "red light" or "green light." We'll do this now with a group of spheres (could be planets, could be electrons) set in orbit around some central attractor. They free-fall towards this attractor, and experience a constant acceleration toward it. All we need to do is keep track of which side of the origin a sphere is on and always apply an acceleration toward the origin.

For reasons that aren't entirely clear, a free-running simulation of this kind tends to accelerate objects to infinite velocities after a while, so to prevent this, a damping factor has been added that reduces each object's velocity by some fractional amount each iteration. This is shown in BUMPZ.BAS. Modulating this damping factor periodically with a sine wave slows all the objects down at the same time, making collections of randomly moving object pause in unison.

```
' BUMPZ.BAS
' Red Light Green Light with Gas Molecules
SCREEN 12
WINDOW (-6.4, -4.8) - (6.4, 4.8)
bls = 9
dist = .5
speed = .01
accel = -.04
damp = .9
DIM embr(bls, bls), d(bls, bls)
                                  ' embrace, distance
                                    ' position
DIM x(bls), y(bls), z(bls)
DIM ox(bls), oy(bls), oz(bls)
                                    ' old position (undraws)
n = 0
   FOR b = 1 TO bls
     x(b) = 10 * (RND - .5)
     y(b) = 10 * (RND - .5)
     z(b) = 10 * (RND - .5)
     ' avoid initial overlap
     FOR c = 1 \text{ TO } b - 1
       d = ((x(b)-x(c)) ^2 + (y(b)-y(c)) ^2 + (z(b)-z(c)) ^2) ^5.5
       ' try again
       IF d < .5 THEN b = b - 1
     NEXT c
     ox(b) = x(b)
     oy(b) = y(b)
     oz(b) = z(b)
     offx(b) = 0 + x(b) - vibrating array animation
     offy(b) = 0 'y(b)
     offz(b) = 0 'z(b)
     vx(b) = (RND - .5) * speed
     vy(b) = (RND - .5) * speed
     vz(b) = (RND - .5) * speed
     FOR c = 1 TO bls
        embr(b, c) = 0
     NEXT c
   NEXT b
DO WHILE INKEY$ = ""
   FOR b = 1 TO bls
```

continued on next page

CHAPTER FIVE

```
continued from previous page
      'position = position + velocity
      x(b) = x(b) + vx(b)
      y(b) = y(b) + vy(b)
      z(b) = z(b) + vz(b)
  NEXT b
  FOR b = 1 TO bls
   ' velocity = velocity + acceleration, centered on the offset
   vx(b) = (vx(b) + accel * SGN(x(b) - offx(b))) * damp
   vy(b) = (vy(b) + accel * SGN(y(b) - offy(b))) * damp
   vz(b) = (vz(b) + accel * SGN(z(b) - offz(b))) * damp
   FOR c = 1 TO bls
         'distance between any two balls
         d(b, c) = ((x(b)-x(c))^2 + (y(b)-y(c))^2 + (z(b)-z(c))^2)^3.5
         ' if two balls hit, swap their velocities
         IF d(b, c) < dist AND embr(b, c) = 0 AND embr(c, b) = 0 THEN
            temp = vx(b)
            vx(b) = vx(c)
            vx(c) = temp
            temp = vy(b)
            vy(b) = vy(c)
            vy(c) = temp
            temp = vz(b)
            vz(b) = vz(c)
            vz(c) = temp
            embr(b, c) = 1
            embr(c, b) = 1
         END IF
        ' embr prevents oscillating collisions:
        ' the balls must first come back apart before checking to see if
        ' they're close enough to swap their velocities again
   IF d(b, c)>dist AND d(c, b)>dist AND embr(b, c)=1 AND embr(c, b)=1 THEN
      embr(b, c) = 0
      embr(c, b) = 0
   END IF
   NEXT c
   CIRCLE (ox(b), oy(b)), dist / 2, 0
   CIRCLE (x(b), y(b)), dist / 2, 12
   ' if you want to see the other axes...
```

```
'CIRCLE (ox(b), oz(b)), dist / 2, 0
'CIRCLE (x(b), z(b)), dist / 2, 13

'CIRCLE (oy(b), oz(b)), dist / 2, 0
'CIRCLE (y(b), z(b)), dist / 2, 14

ox(b) = x(b)
oy(b) = y(b)
oz(b) = z(b)

NEXT b

n = n + 1
damp = .85 + .5 * SIN(n / 2)

LOOP
```

How It Works

BUMPZ.BAS extends the example in Section 5.1 to three dimensions, treating the positions and velocities separately for the x, y, and z axes. Arrays hold values for the control variables for each ball. An initial scattering of balls is generated using a random function, but each new ball is checked to see if it already overlaps one in the field, and if it does, another attempt is made.

The motion of all the balls is towards the origin, set by the variables offx, offy, and offz. We made this an array, one for each ball, so that we could create "mixed fruit vibrating in jello" sort of animation by setting the attractor for each ball to its original position if we want. It was set here to attract all balls to <0,0,0>, but try replacing the zeros with the variables and see what you think.

We display only one view for the coordinates, the x-y plane. We can display the y-z and x-z plane views as well by uncommenting those lines.

The damping is periodically turned up and down using

```
damp = 0.85 + 0.5 * SIN(n / 2)
```

causing it to vary from between 0.35 to 1.35, so that it periodically becomes a gain term, accelerating the motion rather than slowing it.

Polyray Code

One feature that is tough to do inside Polyray is collision detection. Without loops, the only way to check for collisions in Polyray involves writing roughly n^2 IF statements, where n is the number of items. In this case, 9 items would require 81 IF statements. As fun as this might sound, we'll omit it in the Polyray code.

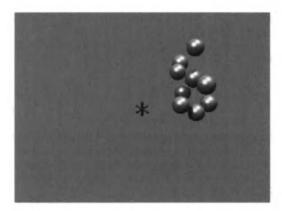


Figure 5-2 Sloshing spheres

The following Polyray code excerpt orbits nine shiny yellow spheres around the origin and modulates all their speeds in a concerted fashion (Figure 5-2).

```
// Orbitals - Particle Systems
start_frame 0
end frame 999
total_frames 1000
outfile plnet
include "\PLY\COLORS.INC"
viewpoint {
   from <25,25,-25>
   at <0,0,0>
   up <0,1,0>
   angle 30
   resolution 160,120
   aspect 1.43
   }
background SkyBlue
light white, <-5,100,-50>
light white, < 5,100, 50>
define dist 0.5
define speed 0.01
define accel -0.04
define damp 1 + 0.5 * SIN(frame / 2)
if (frame == start_frame) {
   // set the initial conditions
```

```
static define bx1 3.16102
   static define by1 7.35870
   static define bz1 5.285980
   static define vx1 -0.00150
   static define vy1 -0.00456
   static define vz1 -0.00410
   static define bx2 0.16661
   static define by 2 3.83032
   static define bz2 6.53091
   static define vx2 -0.00156
   static define vy2 -0.00293
   static define vz2 0.00036
   }
else {
   static define bx1 bx1 + vx1
   static define bx2 bx2 + vx2
   static define bx3 bx3 + vx3
   static define bx4 bx4 + vx4
   static define bx5 bx5 + vx5
   static define bx6 bx6 + vx6
   static define bx7 bx7 + vx7
   static define bx8 bx8 + vx8
   static define bx9 bx9 + vx9
   static define by1 by1 + vy1
   static define by2 by2 + vy2
   static define by3 by3 + vy3
}
// velocity = velocity + acceleration, centered on the origin
if (bx1>0) { static define vx1 vx1 + accel*damp } else { static define vx1 vx1 \Leftarrow
- accel*damp }
if (by1>0) { static define vy1 vy1 + accel*damp } else { static define vy1 vy1 \leftarrow
- accel*damp }
if (bz1>0) { static define vz1 vz1 + accel*damp } else { static define vz1 vz1 \Leftarrow
- accel*damp }
if (bx2>0) { static define vx2 vx2 + accel*damp } else { static define vx2 vx2 ←
- accel*damp }
if (by2>0) { static define vy2 vy2 + accel*damp } else { static define vy2 vy2 ←
- accel*damp }
if (bz2>0) { static define vz2 vz2 + accel*damp } else { static define vz2 vz2 ←
- accel*damp }
if (bx3>0) { static define vx3 vx3 + accel*damp } else { static define vx3 vx3 ←
- accel*damp }
                                                                    continued on next page
```

```
continued from previous page
if (by3>0) { static define vy3 vy3 + accel*damp } else { static define vy3 vy3 ←
- accel*damp }
if (bz3>0) { static define vz3 vz3 + accel*damp } else { static define vz3 vz3 ←
- accel*damp }
object { sphere <bx1,by1,bz1 >,1 shiny_coral }
object { sphere <bx2,by2,bz2 >,1 shiny_coral }
object { sphere <bx3,by3,bz3 >,1 shiny_coral }
object { sphere <bx4,by4,bz4 >,1 shiny_coral }
object { sphere <bx5,by5,bz5 >,1 shiny_coral }
object { sphere <bx6,by6,bz6 >,1 shiny_coral }
object { sphere <bx7,by7,bz7 >,1 shiny_coral }
object { sphere <bx8,by8,bz8 >,1 shiny_coral }
object { sphere <bx9,by9,bz9 >,1 shiny_coral }
object { cylinder < -1, 0, 0>,< 1, 0, 0>,0.1 matte_black }
object { cylinder < 0,-1, 0>,< 0, 1, 0>,0.1 matte_black }
object { cylinder < 0, 0,-1>,< 0, 0, 1>,0.1 matte_black }
```

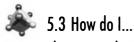
We start by defining the positions (bx, by, and bz) and velocities (vx, vy, and vz) for nine balls. The numbers were generated randomly and plugged into this program using an editor. We use static variables that keep their values from frame to frame, and allow incremental operations (x = x + b) to function. The conditional

```
if (frame == start frame) {
```

lets us set the initial values at the first frame. Afterwards, the positions of the balls will be set by adding their velocity to their position in each frame. We keep a constant acceleration towards the origin with a conditional statement that either adds or subtracts that acceleration from the velocity, depending on which side of the origin we're on.

```
if (bx1>0) { static define vx1 vx1 + accel*damp } else { static define vx1 vx1 \leftarrow - accel*damp }
```

Then we render the spheres and place a small marker at the origin to show its position.



Shatter a cube into fragments?

You'll find the code for this in: PLY\CHAPTER5\SHATTER

Problem

Animations can be created which look like particle systems but which are, in fact, functionally random. The positions of the objects are controlled by formulas with random elements rather than by rules applied randomly to the objects. The distinction here is more syntactic than behavioral. Random diffusion done with particle systems would track the location and momentum of particles perturbed by random functions. A functionally random version of the same animation would dispense with the tracking step and move the particles directly to random locations. In terms of the overall look, they'd be very similar.

Technique

In Superman, The Movie, the criminals on Krypton were placed into some sort of box that went tumbling off into the lesser traveled quadrants of the galaxy where things like HBO and Cinemax are unavailable. This caused them major distress. They eventually escaped, and did massive violence and destruction before the forces of truth, justice, and the American way prevailed.

In the spirit of Hollywood, let's shatter us a cube. The ideas behind the forcing functions that follow were developed by Eric Deren, and, as is the case with all of his code, it's both clever and useful. He exploits the fact that feeding integers to sine and cosine functions basically generates random numbers, and it's this randomness that creates such wonderful scattering. While it's not a real particle systems animation, it sure looks like one.

We define a cube using 12 triangles, with numbered vertices *a*, *b*, and *c*. A cube has 6 sides and we split each side into two triangles each (Figure 5-3). Because 12 triangles don't really scatter nicely, we make 12 more triangles that define a slightly smaller cube hidden inside the first one. We create 24 randomly colored textures and stick them in a file called COL.INC, giving each triangle its own color.

Transforming the Triangles

Each triangle has seven lines of code controlling it. Triangle 1's code is shown here:

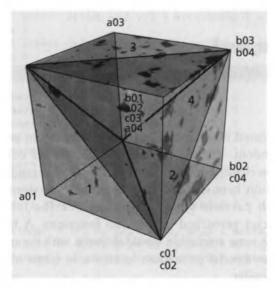


Figure 5-3 A cube done with triangles, with triangles 1-4 shown along with their vertices

```
define fr frame  // keeps the lines shorter

define a01 <-1, 1, 1>
    define b01 < 1, 1, 1>
    define c01 <-1,-1, 1>

define d01 (a01+b01+c01)/3

define e01 < sin(01)*fr*rspeed,sin(01+1)*fr*rspeed,sin(01+2)*fr*rspeed>
    define f01 < sin(01)*fr*lspeed,sin(01+1)*fr*lspeed,sin(01+2)*fr*lspeed>
    define t01 object { polygon 3, a01, b01, c01 translate -d01 rotate e01 
        translate d01+f01 refl_01}
```

The variables a01, b01, and c01 are the vertices. The center of the triangle d01 is found by averaging these vertices. In each frame we feed the triangle's integer identifier (this one is "1") to a sine function that basically generates a random number for that triangle. This number controls the triangle's rotation and displacement. Two scalars, rspeed and lspeed, control the rates for this rotation and displacement.

For an object to be rotated about its center, that center must be at the origin; otherwise, the object gets both translated and rotated. A good analogy is standing on a merry-go-round. If you stand at the center, you spin, anywhere else, you swoop. So, each triangle gets translated to the origin (translated by minus its center or -d01). It's rotated by e01, then translated back to where it was (+d01) plus the additional distance f01, which sends it flying off into space.

Rate Control

Rather than let the departure speed stay constant, we scale it by the frame counter. This makes the cube hang around for a bit, then fly apart quickly. For a ballistic effect, we're firing six spheres at this cube. The triangles rate of departure is made to accelerate at the time the spheres smash into it. Actually, the cube is already breaking up before the spheres arrive, but it's still a nice effect.

Steps

There's really no simple way to automate the creation of the original cube. You have to sit down with a diagram like Figure 5-3 and basically knit the definition by hand. However, the code for shattering it can be automated with the following QuickBasic code.

```
n = 24
OPEN "SHAT.INC" FOR OUTPUT AS #1
FOR x = 1 TO n
   count$ = RIGHT$("00" + LTRIM$(STR$(x)), 2)
   a$ = "a" + count$
   b$ = "b" + count$
   c$ = "c" + count$
   d$ = "d" + count$
   e$ = "e" + count$
   f$ = "f" + count$
   t$ = "t" + count$
   col$ = "refl_" + count$
   PRINT #1, USING "define \ \ (\ \+\ \)/3"; d$, a$, b$, c$
   PRINT #1, USING "define \ \ ←
<\sin(\#\#) *fr*rspeed, \sin(\#\#+1) *fr*rspeed, \sin(\#\#+2) *fr*rspeed>"; e$, x, x, x
   PRINT #1, USING "define \ \ ←
<\sin(\#\#)*fr*lspeed,sin(\#\#+1)*fr*lspeed,sin(\#\#+2)*fr*lspeed>"; f$, x, x, x
   PRINT #1, USING "object { polygon 3,\ \,\ \ \ translate -\ \ rotate \ \ ←
translate \ \+\ \ \
                         \}"; a$, b$, c$, d$, e$, d$, f$, col$
   PRINT #1,
NEXT x
CLOSE #1
           This creates the following output 24 times.
define d01 (a01 +b01 +c01 )/3
define eO1 <sin( 1)*fr*rspeed,sin( 1+1)*fr*rspeed,sin( 1+2)*fr*rspeed>
define f01 <sin( 1)*fr*lspeed,sin( 1+1)*fr*lspeed,sin( 1+2)*fr*lspeed>
object { polygon 3,a01,b01,c01 translate -d01 rotate e01 translate d01+f01 ←
refl_01 }
. . .
```

There's also the matter of generating 24 random colors, one for each triangle. That is handled by the following lines:

```
OPEN "col.inc" FOR OUTPUT AS #1
PRINT #1, "define reflective"
PRINT #1, "surface {"
PRINT #1, " ambient 0.1"
PRINT #1, " diffuse 0.7"
PRINT #1, " specular white, 0.7"
PRINT #1, " microfacet Phong 10"
PRINT #1, " reflection white, 0.7"
PRINT #1, " }"
PRINT #1,
FOR a = 1 TO 24
   x = RND
   y = .5 * RND
   z = RND
   c$ = RIGHT$("00" + LTRIM$(STR$(a)), 2)
   col$ = "color" + c$
   PRINT #1, USING "define \ \ < #.###, #.###, #.### > "; col$, x, y, z
NEXT a
PRINT #1,
FOR a = 1 TO 24
  c$ = RIGHT$("00" + LTRIM$(STR$(a)), 2)
  ref$ = "refl_" + c$
 col$ = "color" + c$
  PRINT #1, USING "define \ \ texture { reflective { color \ \ } }"; ←
ref$, col$
NEXT a
CLOSE #1
           This generates the following include file, COL.INC:
define reflective
surface {
   ambient 0.1
   diffuse 0.7
   specular white, 0.7
   microfacet Phong 10
   reflection white, 0.7
define color01 < 0.7107, 0.4953, 0.8524 >
define color02 < 0.3504, 0.0218, 0.0898 >
define color03 < 0.5111, 0.3777, 0.9354 >
define refl_01 texture { reflective { color color01 } }
define refl_02 texture { reflective { color color02 } }
define refl_03 texture { reflective { color color03 } }
. . .
```

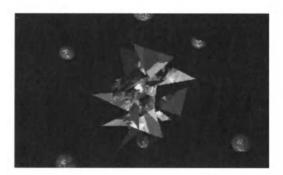


Figure 5-4 Shattering box

Now that all the preliminaries have been completed, the following Polyray data file, SHATTER.PI, creates the animation. It has been shortened to include only the highlights of the program. A sample frame from this animation is shown in Figure 5-4.

```
// shatter.pi
include "..\colors.inc"
start_frame O
end_frame 109
total_frames 110
outfile "shat"
define rspeed frame *0.1
define lspeed frame *0.001
define fr frame
define a01 <-1, 1, 1>
define b01 < 1, 1, 1>
define c01 <-1,-1, 1>
define a02 < 1, 1, 1>
define b02 < 1,-1, 1>
define c02 <-1,-1, 1>
define a03 < 1, 1,-1>
define b03 < 1,-1,-1>
define c03 < 1, 1, 1>
define a04 < 1, 1, 1>
define b04 < 1,-1,-1>
define c04 < 1,-1, 1>
. . .
```

CHAPTER FIVE

```
continued from previous page
define a13 a01*0.9
define b13 b01*0.9
define c13 c01*0.9
define a14 a02*0.9
define b14 b02*0.9
define c14 c02*0.9
define a15 a03*0.9
define b15 b03*0.9
define c15 c03*0.9
define a16 a04*0.9
define b16 b04*0.9
define c16 c04*0.9
include "col.inc"
define d01 (a01+b01+c01)/3
define e01 < sin(01)*fr*rspeed,sin(01+2)*fr*rspeed,sin(01+2)*fr*rspeed>
define f01 < sin(01)*fr*lspeed,sin(01+2)*fr*lspeed,sin(01+2)*fr*lspeed>
define t01 object { polygon 3, a01, b01, c01 translate -d01 rotate e01
translate d01+f01 refl_01}
define d02 (a02+b02+c02)/3
define e02 < sin(02)*fr*rspeed,sin(02+2)*fr*rspeed,sin(02+2)*fr*rspeed>
define f02 < sin(02)*fr*lspeed,sin(02+2)*fr*lspeed,sin(02+2)*fr*lspeed>
define tO2 object { polygon 3, aO2, bO2, cO2 translate -dO2 rotate eO2
translate d02+f02 refl_02}
define d03 (a03+b03+c03)/3
define e03 < \sin(03)*fr*rspeed, \sin(03+2)*fr*rspeed, \sin(03+2)*fr*rspeed>
define f03 < \sin(03)*fr*lspeed, \sin(03+2)*fr*lspeed, \sin(03+2)*fr*lspeed>
define t03 object { polygon 3, a03, b03, c03 translate -d03 rotate e03
translate d03+f03 refl_03}
define d04 (a04+b04+c04)/3
define e04 < sin(04)*fr*rspeed,sin(04+2)*fr*rspeed,sin(04+2)*fr*rspeed>
define f04 < \sin(04)*fr*lspeed, \sin(04+2)*fr*lspeed, \sin(04+2)*fr*lspeed>
define t04 object { polygon 3, a04, b04, c04 translate -d04 rotate e04
translate d04+f04 refl_04}
// Polyray's parser doesn't care if you call one item per line
// or all at once. These are the triangle objects
   t01 t02 t03 t04 t05 t06 t07 t08 t09 t10 t11 t12
   t13 t14 t15 t16 t17 t18 t19 t20 t21 t22 t23 t24
// set up background color & lights
background midnightblue
```

```
light <0,15,-12>
light <12,-15,-12>
light <2,2,2>,<0,0,0>
define bumpy_yellow
texture {
   special surface {
      color <1.5,0.8,0>
      normal N + (dnoise(3*W) - white/2)
      ambient 0.2
      diffuse 0.3
      specular white, 0.7
      microfacet Cook 5
   scale <0.05, 0.05, 0.05>
object {sphere <-10, 0, 0>,0.4 translate <frame/2,0,0> bumpy_yellow}
object {sphere < 0,-10, 0>,0.4 translate <0,frame/2,0> bumpy_yellow}
object {sphere < 0, 0,-10>,0.4 translate <0,0,frame/2> bumpy_yellow}
object {sphere < 10, 0, 0>,0.4 translate <-frame/2,0,0> bumpy_yellow}
object {sphere < 0, 10, 0>,0.4 translate <0,-frame/2,0> bumpy_yellow}
object {sphere < 0, 0, 10>,0.4 translate <0,0,-frame/2> bumpy_yellow}
viewpoint {
   from <6, 12,-8>
   at <0.0.0>
      <0, 1, 0>
   angle 30
   resolution 80,50
   aspect 1.43
```

How It Works

This abbreviated listing for SHATTER.PI shows the definitions of the triangles (big and little) and the triangle dispersal control code which sends our triangles flying off. We create a bumpy yellow texture, and apply it to six spheres symmetrically arranged around the origin. They all approach the origin simultaneously and collide with the box at frame 20. We've multiplied both the rotational speed and the leaving speed of the triangles by the frame counter

```
define rspeed frame*0.1 define lspeed frame*0.001
```

so both elements accelerate as the animation progresses. Scaling them by 0.1 and 0.001 coordinates the motions, making the box fly apart just as the balls arrive to smash it.



You'll find the code for this in: PLY\CHAPTER5\SHATBLOB

Problem

There are two distinct ways that objects are defined in ray tracers. Objects like spheres and blobs have functional definitions, meaning equations are solved to define their form. They have smooth surfaces even under close inspection. Everything else is usually just a collection of triangles. It's not easy to morph a functional object into one based on triangles. The "to" and "from" shapes have to both be functional or both be triangle-based for it to work at all. Shattering a functionally based object is out of the question. The trick is to convert it into a triangle-based one and then hit it with Eric Deren's hammer.

Technique

A very useful addition that Alexander Enzmann made to Polyray was triangle output. It makes it possible to play with functionally defined objects like blobs as triangle meshes. Making the standard tricornered blob object that comes with Polyray into a collection of triangles is as simple as typing:

\ply\polyray blob.pi -t 0 -r 3 -V 0 >blob.raw

We can then disperse it using the cube shattering code in Section 5.3. The raw output is simply a series of three x, y, and z coordinates defining the vertices for the triangles. We use a QuickBasic program to convert these into an acceptable Polyray data file, and then another program to automate writing the individual control code for each triangular piece.

Steps

We will create a Polyray file that fragments a blob into 828 triangles. We start with the file BLOB.RAW created using the triangle output option in Polyray and use QuickBasic code to convert it into a format acceptable for rendering. We also need to generate a random collection of textures, and have it create the dispersal code similar to the code in the last example, only capable of handling more objects:

```
OPEN "blob.raw" FOR INPUT AS #1
OPEN "shatblob.pi" FOR OUTPUT AS #2
CLS
```

```
PRINT #2, "// shatblob.pi"
PRINT #2, ""
PRINT #2, "start_frame O"
PRINT #2, "end_frame 180"
PRINT #2, "total_frames 181"
PRINT #2,
PRINT #2, "outfile bshat"
PRINT #2,
PRINT #2, "define fr frame-90"
PRINT #2, "define rspeed fr*0.1"
PRINT #2, "define lspeed fr*0.001"
PRINT #2,
PRINT #2, "viewpoint {"
PRINT #2, " from <0,0,-5>"
PRINT #2, "
            at <0,0,0>"
PRINT #2, " up <0,1,0>"
PRINT #2, " angle 30"
PRINT #2, "
             resolution 320,240"
PRINT #2, "
             aspect 1.33"
PRINT #2, " }"
PRINT #2, "background SkyBlue"
PRINT #2, "light 0.6 * white, <-15,30,-25>"
PRINT #2, "light 0.6 * white, < 15,30,-25>"
PRINT #2,
n = 1
' read and format the RAW triangle file
PRINT #2, "// triangles"
DO WHILE NOT EOF(1)
   INPUT #1, x1, y1, z1, x2, y2, z2, x3, y3, z3
   INPUT #1, nx1, ny1, nz1, nx2, ny2, nz2, nx3, ny3, nz3
   INPUT #1, u1, v1, u2, v2, u3, v3
   count$ = RIGHT$("000" + LTRIM$(STR$(n)), 3)
   a$ = "a" + count$
   b$ = "b" + count$
   c$ = "c" + count$
   PRINT #2, USING "define \ \ <###.####,###.####,###.###">"; a$, x1, y1, z1
   PRINT #2, USING "define \ \ <###.####,###.####,###.###">"; b$, x2, y2, z2
   PRINT #2, USING "define \ \ <###.####,###.####,###."; c$, x3, y3, z3
   PRINT #2,
   n = n + 1
Locate 1,1: PRINT'Processing Triange"; n-1
L<sub>00</sub>P
CLOSE #1
n = n - 1
' create our random colors, one per triangle
PRINT #2, "define reflective"
                                                                  continued on next page
```

```
continued from previous page
PRINT #2, "surface {"
PRINT #2, " ambient 0.1"
PRINT #2, " diffuse 0.7"
PRINT #2, " specular white, 0.7"
PRINT #2, " microfacet Phong 10"
PRINT #2, " reflection white, 0.7"
PRINT #2, " }"
PRINT #2,
PRINT #2, "// textures"
PRINT #2,
FOR a = 1 TO n
   r = RND
  g = .5 * RND
  b = RND
   count$ = RIGHT$("000" + LTRIM$(STR$(a)), 3)
   tex$ = "refl_" + count$
   PRINT #2, USING "define \
                             \ texture { reflective { color <#.####, ←
#.####, #.#### > } }"; tex$, r, g, b
NEXT a
PRINT #2,
' create the Triangle Dispersion Code, "Eric's Hammer"
PRINT #2, "// Eric's Hammer"
PRINT #2,
FOR x = 1 TO n
   count$ = RIGHT$("000" + LTRIM$(STR$(x)), 3)
   a$ = "a" + count$
   b$ = "b" + count$
   c$ = "c" + count$
   d$ = "d" + count$
   e$ = "e" + count$
   f$ = "f" + count$
   t$ = "t" + count$
   col$ = "refl_" + count$
   PRINT #2, USING "define \ \ (\ \+\ \)/3"; d$, a$, b$, c$
   PRINT #2, USING "define \ \ < ←
sin(###)*fr*rspeed,sin(###+1)*fr*rspeed,sin(###+2)*fr*rspeed>"; e$, x, x, x
   PRINT #2, USING "define \ \ < ←
sin(###)*fr*lspeed,sin(###+1)*fr*lspeed,sin(###+2)*fr*lspeed>"; f$, x, x, x
   PRINT #2, USING "object { polygon 3, \ \, \ \ translate -\ \ ←
rotate \ \ translate \ \+\ \ \
                                       \}"; a$, b$, c$, d$, e$, d$, f$, col$
   PRINT #2,
NEXT x
CLOSE #2
```

The file that this code generates has four basic sections. There is the standard header specifying things like the viewpoint and the frame count, the triangles (828 of them), the textures (828 of them), and the dispersal code:

```
// shatblob.pi
start_frame O
end_frame 180
total_frames 181
outfile bshat
define fr frame-90
define rspeed fr*0.1
define lspeed fr*0.001
viewpoint {
   from <0,0,-5>
   at <0,0,0>
   up <0,1,0>
   angle 30
   resolution 320,240
   aspect 1.33
background SkyBlue
light 0.6 * white, <-15,30,-25>
light 0.6 * white, < 15,30,-25>
// triangles
define a001 < -0.71540, -0.82300, -0.00705>
define b001 < -0.71540, -0.78140, -0.03992>
define c001 < -0.73560, -0.78140, -0.00705>
define a002 < -0.71540, -0.90870, 0.16630>
define b002 < -0.73560, -0.78140, -0.00705>
define c002 < -0.77690, -0.78140, 0.16630>
define a003 < -0.71540, -0.90870, 0.16630>
define b003 < -0.71540, -0.82300, -0.00705>
define c003 < -0.73560, -0.78140, -0.00705>
. . .
define reflective
surface {
   ambient 0.1
   diffuse 0.7
   specular white, 0.7
   microfacet Phong 10
   reflection white, 0.7
   }
// textures
```

```
continued from previous page
define refl_001 texture { reflective { color <0.71073, 0.49529, 0.85240 > } }
define refl_002 texture { reflective { color <0.35038, 0.02182, 0.08978 > } }
define refl_003 texture { reflective { color <0.51111, 0.37768, 0.93539 > } }
. . .
// dispersal code
define d001 (a001+b001+c001)/3
define e001 < sin( 1)*fr*rspeed,sin( 1+1)*fr*rspeed,sin( 1+2)*fr*rspeed>
define f001 < sin( 1)*fr*lspeed,sin( 1+1)*fr*lspeed,sin( 1+2)*fr*lspeed>
object { polygon 3, a001, b001, c001 translate -d001 rotate e001 translate \Leftarrow
d001+f001 refl_001 }
define d002 (a002+b002+c002)/3
define e002 < sin( 2)*fr*rspeed,sin( 2+1)*fr*rspeed,sin( 2+2)*fr*rspeed>
define f002 < sin( 2)*fr*lspeed,sin( 2+1)*fr*lspeed,sin( 2+2)*fr*lspeed>
object { polygon 3, a002, b002, c002 translate -d002 rotate e002 translate \Leftarrow
d002+f002 refl_002 }
define d003 (a003+b003+c003)/3
define e003 < sin( 3)*fr*rspeed,sin( 3+1)*fr*rspeed,sin( 3+2)*fr*rspeed>
define f003 < sin( 3)*fr*lspeed,sin( 3+1)*fr*lspeed,sin( 3+2)*fr*lspeed>
object { polygon 3, a003, b003, c003 translate -d003 rotate e003 translate ←
d003+f003 refl_003 }
. . .
. . .
```

How It Works

We decided to start here with a shattered blob, bring it back together, then fly it apart again. Looking at the dispersion code above, the variable fr controls the amount of scattering. When fr = 0, we have our fully assembled object. Any value other than zero leads to a dismantled object. We'll use a 180 frame animation, and the following statement.

```
define fr frame - 90
```

This line means that at frame 0, fr = -90 and our blob is shattered out of view. It assembles itself at frame 90, (where fr = 0) and flies apart again by frame 180 (fr = 90). The variables *lspeed* and *rspeed* scale how far fr = -90 takes us out, and the rate of rotation of the pieces as they come and go. It's shown flying apart in Figure 5-5.

Comments

One of the problems dealing with files like these (this one's over 400K) is how long it takes to load into an editor. Sometimes all you want to do is adjust something small like the viewpoint or the lighting. It's easier to split

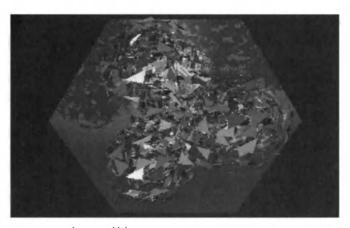


Figure 5-5 Shattering blob

this file in two, creating a small control file containing just the viewpoint, lights, and constants (SHATBLB.PI) and a large data file (FRAGBLOB.INC), which contains the traingles, the textures and the motion code. This larger file is included in the control file using the line

include "fragblob.inc"

Another problem you'll encounter with files of this size is long preprocessing time (over a minute per frame for this file). In developing control structures for your own animations, you may want to reduce the item count to under 100 during development, which will preprocess quickly and allow you to adjust the motion to suit you. Once you're happy, then increase it to the full count for the final render.



5.5 How do I... Give a salt crystal indigestion?

You'll find the code for this in: PLY\CHAPTER5\TUG

Problem

Let's say you have a cubical crystal lattice and you'd like to make it appear that something was crawling around inside it, forcing the lattice to bulge out and distort as a result of this internal perturbation. Once the disturbance passed, the lattice would return to its original form. This effect can be created by defining the position and radius of the spheres in the lattice to vary depending on how close they are to repellers orbiting freely throughout the structure.

Technique

Crystalline structures are highly ordered collections of objects in 3-D space. This animation starts with such a structure, a six by six by six array of spheres. Small light sources orbit this array and perturb both the positions and (later on) the diameters of the spheres proportionally to their distance from the spheres (see Figure 5-6).

Since light sources are invisible in ray tracers, we've placed spheres around them to make them visible. We turn off their shadows so the light can get through. Discs surrounding our structure will catch the shadows as both the lights move around and the entire shape changes structure. We'll place the camera on an orbital platform and circle the structure for a complete view of the proceedings.

How It Works

Rather than give the animation code first and then explain how it works, we're going to cover what's being done first, and then detail how certain effects were achieved.

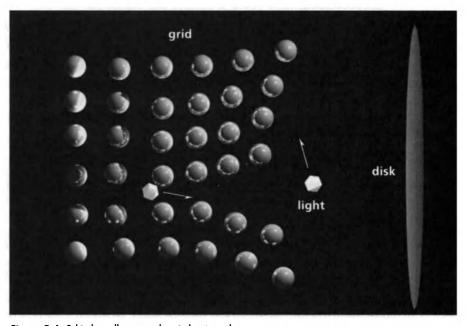


Figure 5-6 Orbital repellers perturb a six by six grid

The orbital lights are the central controlling elements in this animation. They are placed into circular motions using sines and cosines, and offset using constants and phases:

```
define ph1 0*pi/3
define ph2 2*pi/3
define ph3 4*pi/3

// three little orbital repellers

define fx1 2.5 + 5 * COS(angle_nor+ph1)
define fy1 2.5 + 5 * SIN(angle_nor+ph1)
define fz1 2.5

define fx2 2.5 + 5 * COS(angle_nor+ph2)
define fy2 2.5
define fx2 2.5 + 5 * SIN(angle_nor+ph2)
define fx3 2.5
define fx3 2.5
define fy3 2.5 + 5 * COS(angle_nor+ph3)
define fx3 2.5 + 5 * SIN(angle_nor+ph3)
```

The grid is built using QuickBasic code. The positions of every object must allow these orbital elements to influence their position and size. The variables a, b, and c are the array counters for the grid, and also the positions of the grid elements when they're at rest. The first element (0,0,0) is defined using the following lines

```
define a 00
define b 00
define c 00
include "tug.inc"

define x000000 a + f1 * drx1 + f2 * drx2 + f3 * drx3
define y000000 b + f1 * dry1 + f2 * dry2 + f3 * dry3
define z000000 c + f1 * drz1 + f2 * drz2 + f3 * drz3
object { sphere <x000000,y000000,z000000>,0.2 reflective_white }
```

The TUG.INC file generates the factors, based on where the orbital lights are, that shift the positions of the elements in the grid. This code is used for each sphere in the grid. With 216 objects, making it a single include file saved a lot of disk space. The repeated file TUG.INC contains the following code:

```
define d1 ((a - fx1) ^2 + (b - fy1) ^2 + (c - fz1) ^2) ^0.5
define d2 ((a - fx2) ^2 + (b - fy2) ^2 + (c - fz2) ^2) ^0.5
define d3 ((a - fx3) ^2 + (b - fy3) ^2 + (c - fz3) ^2) ^0.5
define drx1 a - fx1
```

```
continued from previous page

define dry1 b - fy1

define drz1 c - fz1

define drx2 a - fx2

define dry2 b - fy2

define drz2 c - fz2

define drx3 a - fx3

define dry3 b - fy3

define dry3 c - fz3

define f1 f / (1 + d1 ^ p)

define f2 f / (1 + d2 ^ p)

define f3 f / (1 + d3 ^ p)
```

The variables a, b, and c are the original undisturbed positions of the spheres in the grid. The distance between them and the orbital lights is calculated as d1, d2 and d3, one for each light. The direction from the sphere to the light sphere is given by $drx\ dry$ and drz. The magnitude of the resulting force, modeled to resemble electrostatic repulsion, was then calculated as the reciprocal of distance squared (1/dist²), as f1, f2, and f3. The positions of the spheres are then calculated as their base position plus the force times its direction:

```
define x000000 a + f1 * drx1 + f2 * drx2 + f3 * drx3
define y000000 b + f1 * dry1 + f2 * dry2 + f3 * dry3
define z000000 c + f1 * drz1 + f2 * drz2 + f3 * drz3
```

Occasionally, the distance between the light and the sphere can be zero, resulting in an infinite force. The blinding flash will reduce your monitor to a glowing mound of slag (only kidding, folks). You won't like that. We bump the value up by 1, and keep civilization safe.

```
define f1 f / (1 + d1 ^ p)
define f2 f / (1 + d2 ^ p)
define f3 f / (1 + d3 ^ p)
```

The variable p equals 2, but it can be changed to experiment with various force relationships between the lights and the spheres.

Polyray Code

We'll break this project down into three separate animations. The first animation, TUG1.PI, holds the diameters of the spheres grid constant as in Figure 5-6. The orbital lights stay outside the array for the most part. The main feature of this animation is the intensely colored shadows the array projects onto discs we've placed around the figure. These disks not only provide shadow clues, their reflections off the spheres make them appear textured. This continues for a total of 216 spheres.

```
//TUG1.PI
start_frame O
end_frame 179
total_frames 180
outfile "tug1"
include "colors.inc"
viewpoint {
   from <10,15,-10>
   at <2.5,2.5,2.5>
   up <0,1,0>
   angle 30
   resolution 320,200
   aspect 1.43
   }
background SkyBlue
define pi 3.14159
define angle_nor frame*2*pi/total_frames
// little orbital repellers
define fx1 2.5 + 5 * COS(angle_nor)
define fy1 2.5 + 5 * SIN(angle_nor)
define fz1 2.5
define fx2 2.5 + 5 * COS(angle_nor)
define fy2 2.5
define fz2 2.5 - 5 * SIN(angle_nor)
define fx3 2.5
define fy3 2.5 + 5 * SIN(angle_nor)
define fz3 2.5 - 5 * COS(angle_nor)
// unused 4th repeller - could add more
define fx4 0
define fy4 0
define fz4 0
define f 1
define p 2
object {
   sphere \langle fx1, fy1, fz1 \rangle, 0.2
   shading_flags 32 + 8 + 4 + 2 + 1
   shiny_yellow
}
object {
```

```
continued from previous page
   sphere <fx2,fy2,fz2>,0.2
   shading_flags 32 + 8 + 4 + 2 + 1
   shiny_blue
}
object {
   sphere <fx3,fy3,fz3>,0.2
   shading_flags 32 + 8 + 4 + 2 + 1
   shiny_red
}
light <0.8,0.8,0>, <fx1,fy1,fz1>
light <0,0,0.8>, <fx2,fy2,fz2>
light <0.8,0,0>, <fx3,fy3,fz3>
light white*0.5, <9,15,-10>
object { disc <-20,0,0>, <1,0,0>, 20 matte_white }
object { disc <0,-20,0>, <0,1,0>, 20 matte_white }
object { disc <0,0,-20>, <0,0,1>, 20 matte_white }
object { disc <20,0,0>, <1,0,0>, 20 matte_white }
object { disc <0,20,0>, <0,1,0>, 20 matte_white }
object { disc <0,0,20>, <0,0,1>, 20 matte_white }
main loop
define a 00
define b 00
define c 00
include "tug.inc"
define x000000 a + f1 * drx1 + f2 * drx2 + f3 * drx3 + f4 * drx4
define y000000 b + f1 * dry1 + f2 * dry2 + f3 * dry3 + f4 * dry4
define z000000 c + f1 * drz1 + f2 * drz2 + f3 * drz3 + f4 * drz4
object { sphere <x000000,y000000,z000000>,0.2 matte_white }
define a 00
define b 00
define c 01
include "tug.inc"
define x000001 a + f1 * drx1 + f2 * drx2 + f3 * drx3 + f4 * drx4
define y000001 b + f1 * dry1 + f2 * dry2 + f3 * dry3 + f4 * dry4
define z000001 c + f1 * drz1 + f2 * drz2 + f3 * drz3 + f4 * drz4
object { sphere <x000001,y000001,z000001>,0.2 matte_white }
```

This animation employs a fixed vantage point and three orbital lights in the x, y and z plane, centered more or less on the center of our cube. A sample frame is shown in Figure 5-7.

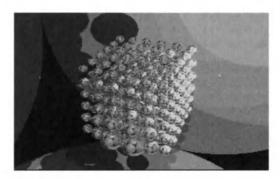


Figure 5-7 Sample from TUG1.PI

This animation was good, but for the sake of more interaction, let's move the locations of the forcing elements two times closer to the center of the grid than they were in TUG1.PI. This is only for the force calculations. In TUG2.PI, the lights and the spheres stay external and show up on the outside, whereas the forcing elements are on the inside of the array. This continues on for a total of 216 spheres.

```
// TUG2.PI External Lights - Internal Forces
start_frame O
end_frame 143
total_frames 144
outfile "tug"
include "\PLY\COLORS.INC"
define pi 3.14159
define angle_nor frame*2*pi/total_frames
viewpoint {
   from rotate(<10,10+5*cos(angle_nor),~15>,<0,360*angle_nor/(2*pi),0>)
   at <2.5,2.5,2.5>
   up <0,1,0>
   angle 30
   resolution 320,200
   aspect 1.43
//background SkyBlue
define ph1 0*pi/3
define ph2 2*pi/3
define ph3 4*pi/3
// little orbital repellers
define orbit 2
```

```
continued from previous page
define fx1 3 + orbit * COS(angle_nor * 4 + ph1)
define fy1 3 + 1.5 * COS(angle_nor) + orbit * SIN(angle_nor * 4 + ph1)
define fz1 3
define fx2 3 + 1.5 * COS(angle_nor) + orbit * COS(angle_nor * 4 + ph2)
define fy2 3
define fz2 3 + orbit * SIN(angle_nor * 4 + ph2)
define fx3 3
define fy3 3 + orbit * COS(angle_nor * 4 + ph3)
define fz3 3 + 1.5 * COS(angle_nor) + orbit * SIN(angle_nor * 4 + ph3)
// UNUSED 4th Repeller
define fx4 0
define fy4 0
define fz4 0
define f 1
define p 2
object {
   sphere <2*fx1,2*fy1,2*fz1>,0.2
   shading_flags 32 + 8 + 4 + 2 + 1
   shiny_yellow
}
object {
   sphere <2*fx2,2*fy2,2*fz2>,0.2
   shading_flags 32 + 8 + 4 + 2 + 1
   shiny_blue
}
object {
   sphere <2*fx3,2*fy3,2*fz3>,0.2
   shading_flags 32 + 8 + 4 + 2 + 1
   shiny_red
}
light <0.8,0.8,0>, <2*fx1,2*fy1,2*fz1>
light <0,0,0.8>, <2*fx2,2*fy2,2*fz2>
light <0.8,0,0>, <2*fx3,2*fy3,2*fz3>
light white*0.25, <9,15,-10>
object { disc <-20,0,0>,<1,0,0>, 20 matte_white }
object { disc <0,-20,0>,<0,1,0>, 20 matte_white }
object { disc <0,0, 20>,<0,0,1>, 20 matte_white }
object { disc <20,0,0>,<1,0,0>, 20 matte_white }
object { disc <0,20,0>,<0,1,0>, 20 matte_white }
object { disc <0,0,-20>,<0,0,1>, 20 matte_white }
```

```
main loop
define a 00
define b 00
define c 00
include "tug.inc"
define x000000 a + f1 * drx1 + f2 * drx2 + f3 * drx3
define y000000 b + f1 * dry1 + f2 * dry2 + f3 * dry3
define z000000 c + f1 * drz1 + f2 * drz2 + f3 * drz3
object { sphere <x000000,y000000,z000000>,0.2+f1+f2+f3 reflective_white }
define a 00
define b 00
define c 01
include "tug.inc"
define x000001 a + f1 * drx1 + f2 * drx2 + f3 * drx3
define y000001 b + f1 * dry1 + f2 * dry2 + f3 * dry3
define z000001 c + f1 * drz1 + f2 * drz2 + f3 * drz3
object { sphere <x000001,y000001,z000001>,0.2+f1+f2+f3 reflective white }
```

Here we're moving around the viewpoint and making the lights move in more complex patterns. For example, the f3 light moves in the y-z plane as shown in Figure 5-8. The motion of the forcing element has been reduced so that it stays mostly within the confines of the grid (represented by a box in Figure 5-8), while the light motion, being this path multiplied by two, stays mostly outside the grid:

```
light <0.8,0,0>, <2*fx3,2*fy3,2*fz3>
```

We've also made it so that the diameters of the spheres in the grid are influenced by the sum of the forces acting on them, so that when a light approaches a sphere, it both grows larger and moves away:

```
object { sphere <x000000,y000000,z000000>,0.2+f1+f2+f3 reflective_white }
```

This makes our mass of spheres billow as lights pass by them (Figure 5-9). An example of this file's output is shown in Figure 5-10.

For completeness, the last animation (TUG3.PI) moves both the forcing functions and the spheres and lights inside the array. It's a dark animation, since most of the time the lights are lost somewhere in the array, but occasionally light comes bursting out of the array like a salt crystal with indigestion. One such flash is shown in Figure 5-11.

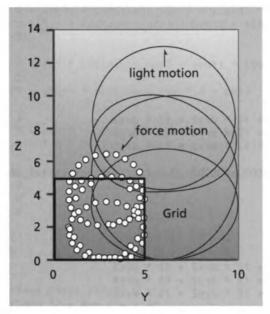


Figure 5-8 Orbital motions of the light and the displacing force in TUG2

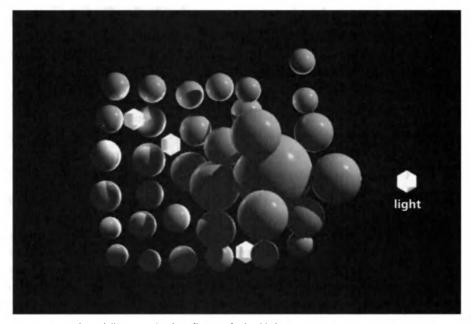


Figure 5-9 Spheres billowing under the influence of orbital lights

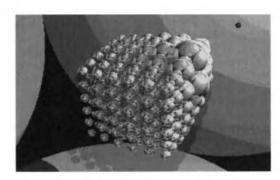


Figure 5-10 TUG2.PI sample image

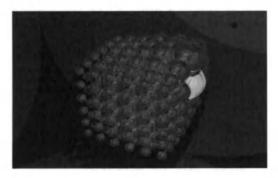


Figure 5-11 Salt crystal with indigestion

```
// TUG3.PI Internal Lights - Internal Forces
start_frame 0
end_frame 143
total_frames 144
outfile "tug"
include "\PLY\COLORS.INC"
define pi 3.14159
define angle_nor frame*2*pi/total_frames
viewpoint {
   from rotate(<10,10+5*cos(angle_nor),-15>,<0,360*angle_nor/(2*pi),0>)
   at <2.5,2.5,2.5>
   up <0,1,0>
   angle 30
   resolution 320,200
   aspect 1.43
define ph1 0*pi/3
```

CHAPTER FIVE

```
continued from previous page
define ph2 2*pi/3
define ph3 4*pi/3
// little orbital repellers
define orbit 2
define fx1 3 + orbit * COS(angle_nor * 4 + ph1)
define fy1 3 + 1.5 * COS(angle_nor) + orbit * SIN(angle_nor * 4 + ph1)
define fz1 3
define fx2 3 + 1.5 * COS(angle_nor) + orbit * COS(angle_nor * 4 + ph2)
define fy2 3
define fz2 3 + orbit * SIN(angle_nor * 4 + ph2)
define fx3 3
define fy3 3 + orbit * COS(angle_nor * 4 + ph3)
define fz3 3 + 1.5 * COS(angle_nor) + orbit * SIN(angle_nor * 4 + ph3)
// UNUSED 4th repeller
define fx4 0
define fy4 0
define fz4 0
define f 1
define p 2
object {
   sphere \langle fx1, fy1, fz1 \rangle, 0.2
   shading_flags 32 + 8 + 4 + 2 + 1
   shiny_yellow
}
object {
   sphere \langle fx2, fy2, fz2 \rangle, 0.2
   shading_flags 32 + 8 + 4 + 2 + 1
   shiny_blue
}
object {
   sphere <fx3,fy3,fz3>,0.2
   shading_flags 32 + 8 + 4 + 2 + 1
   shiny_red
}
light <0.8,0.8,0>, <fx1,fy1,fz1>
light <0,0,0.8>, <fx2,fy2,fz2>
light <0.8,0,0>, <fx3,fy3,fz3>
light white*0.25, <9,15,-10>
object { disc <-20,0,0>,<1,0,0>, 20 matte_white }
object { disc <0,-20,0>,<0,1,0>, 20 matte_white }
```

```
object { disc <0,0, 20>,<0,0,1>, 20 matte_white } object { disc <20,0,0>,<1,0,0>, 20 matte_white } object { disc <0,20,0>,<0,1,0>, 20 matte_white } object { disc <0,0,-20>,<0,1,>, 20 matte_white }
```

Main Loop

As with all animations involving several hundred objects, we automate the creation of the 216 elements programmatically. In this particular case, we can also place repetitive code in an include file, call it as needed, and reduce the file size:

```
PRINT #1, "//main loop"
PRINT #1,
FOR a = 0 TO 5
FOR b = 0 TO 5
FOR c = 0 TO 5
a$ = RIGHT$("00" + LTRIM$(STR$(a)), 2)
b$ = RIGHT$("00" + LTRIM$(STR$(b)), 2)
c$ = RIGHT$("00" + LTRIM$(STR$(c)), 2)
x$ = "x" + a$ + b$ + c$
y$ = "y" + a$ + b$ + c$
z$ = "z" + a$ + b$ + c$
PRINT #1, "define a "; a$
PRINT #1, "define b "; b$
PRINT #1, "define c "; c$
PRINT #1,
PRINT #1, "include "; CHR$(34); "tug.inc"; CHR$(34)
PRINT #1,
PRINT #1, "define "; x$; " a + f1 * drx1 + f2 * drx2 + f3 * drx3"
PRINT #1, "define "; y$; " b + f1 * dry1 + f2 * dry2 + f3 * dry3"
PRINT #1, "define "; z$; " c + f1 * drz1 + f2 * drz2 + f3 * drz3"
PRINT #1,
PRINT #1, USING "object { sphere <\
                                     \,\\\,\\\>,0.2+f1+f2+f3
reflective__white }"; x$, y$, z$
NEXT c
NEXT b
NEXT a
```

We convert the variables *a*, *b*, and *c* to strings, join them together to create our position control variable names, and then loop through and automatically create all the control code for the objects. This process could be extended to generate thousands of elements, although we'd eventually run out of memory and the patience to render such a scene.



5.6 How do I...

Create a swarming swirling twisting snake-like mass that goes on forever?

You'll find the code for this in: PLY\CHAPTER5\SWARM

Problem

If you're interested in generating swirling masses of coordinated objects like those common to many popular Windows screen saver programs, it's possible to extend their 2-D representations to 3-D and use blobs to connect various parts into a swirling array. Take for example Eric Deren's program MERGE2. It generates a motion popular in many screen saver programs, where a collection of objects flock and twist around each other. Eric posted this on the Leo BBS, it migrated to the TGA BBS, and Eric was generous enough to supply his source code. Code swapping is one of the great benefits provided by BBSs, allowing information to migrate and spark collective creativity. His code was extended to 3-D, formatted for Polyray, and within a week became the following animation. It's all sines and cosines, but the motion is hypnotic. It will repeat about every 7000 frames.

Technique

This code creates a swirling swarm of 60 blobs defined by 300 metaballs. Some examples of individual frames, with the blobs represented as single lines, are shown in Figure 5-12.

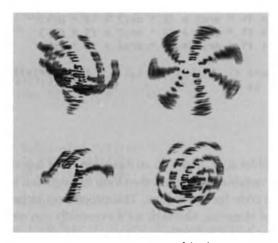


Figure 5-12 Swarm in various stages of development

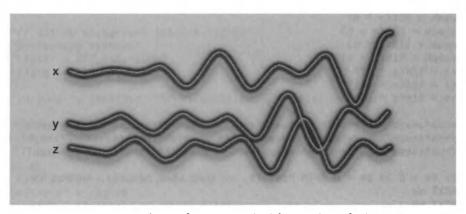


Figure 5-13 Overtones in complex waveforms generate the shifting coordinates for this animation

The Motion

The motion results from the combination of overtones in a series of sine waves. An illustration of the waveforms used to generate this motion is shown in Figure 5-13.

These waveforms are generated using combinations of sines and cosines, where the controlling factors are integers, generating a waveform with seemingly random shifting harmonics. Because these numbers are regularly indexed down the wave form, however, the motion created has a certain fluid grace about it.

A fast running simulation program was written in C, and the executable is included on the disc. However, as this is a QuickBasic book, a slower QuickBasic version (SEESWARM.BAS) is given here. Individual blob elements are generated and combined in sets of five to produce elongated forms.

```
' SEESWARM.BAS
'swarm simulation code

TRAIL = 4
ITEMS = 60
SCREEN 12
'WINDOW SCREEN (0, 0)-(639, 479)
ampl = 45

DO WHILE INKEY$ = ""
frame = frame + 1
FOR bb = 0 TO ITEMS
b = 1.5 + bb / 33.33333333#

FOR aa = 0 TO TRAIL
a = (aa + frame) / 33.333333333#
```

```
continued from previous page
csab = COS(a * b)
csapb = COS(a + b)
snab = SIN(a * b)
snapb = SIN(a + b)
sa = SIN(a * .1)
ca = COS(a * .1)
s5 = SIN(a * .05)
x=sa*snapb*ampl*csab+ca*(csapb*ampl*b*csab+s5*(ampl*(csab+2*snapb))) + 320
y=sa*csapb*ampl*csab+ca*(csapb*ampl*b*snab+s5*(ampl*(cssab+2*csapb))) + 240
z=ca*csapb*ampl*csab+ca*(snapb*ampl*b*snab+s5*(ampl*(snab+2*snapb))) + 200
IF aa = 0 OR bb = 0 THEN PSET (x, y) ELSE LINE -(x, y), bb MOD 15
NEXT aa
NEXT bb
CLS
L00P
```

Writing the Program

The Polyray data file SWARM.PI needed to generate this animation, is enormous, although the QuickBasic code is fairly short. SWARM.BAS is as follows.

```
OPEN "swarm.pi" FOR OUTPUT AS #1
PRINT #1, "//"
PRINT #1, "// SWARM.PI"
PRINT #1, "//"
PRINT #1, "// Polyray input file - Jeff Bowermaster"
PRINT #1, "// Based on MERGE2 by Eric Deren
PRINT #1,
PRINT #1, "start_frame
PRINT #1, "end_frame
                       3500"
PRINT #1, "total_frames 3501"
PRINT #1, "outfile "; CHR$(34); "swrm"; CHR$(34)
PRINT #1,
PRINT #1, "define min 2.0"
PRINT #1, "define str 4.0"
PRINT #1, "define radius 15.0"
PRINT #1, "define u 20"
PRINT #1, "define v 20"
PRINT #1,
PRINT #1, "// set up the camera"
PRINT #1, "viewpoint {"
PRINT #1, " from <0,100,-225>"
PRINT #1, " at <0,0,0>"
PRINT #1, " up <0,1,0>"
PRINT #1, " angle 45"
PRINT #1, " resolution 160,120"
PRINT #1, " aspect 1.333"
PRINT #1, " }"
```

```
PRINT #1,
PRINT #1, "// set up background color & lights"
PRINT #1, "background skyblue"
PRINT #1, "light < 220,240,-800>"
PRINT #1, "light < 420,240,-800>"
PRINT #1,
PRINT #1, "include "; CHR$(34); "\PLY\COLORS.INC"; CHR$(34)
PRINT #1,
PRINT #1, "define ampl 45"
PRINT #1,
FOR x = 0 TO 4
  count$ = RIGHT$("00" + LTRIM$(STR$(x)), 2)
  var$ = "define a" + count$
  PRINT #1, var$; " (frame+"; x; ")/33.333333"
NEXT x
PRINT #1,
FOR x = 0 TO 59
  count = RIGHT ("00" + LTRIM (STR(x)), 2)
  var$ = "define b" + count$
  PRINT #1, var$;
  PRINT #1, USING " #.## "; 1.5 + x / 33.333333#
NEXT x
PRINT #1,
FOR y = 0 TO 4
FOR x = 0 TO 59
   counta$ = RIGHT$("00" + LTRIM$(STR$(y)), 2)
   countb$ = RIGHT$("00" + LTRIM$(STR$(x)), 2)
   countc$ = counta$ + countb$
PRINT #1, USING "define a a\\"; counta$
PRINT #1, USING "define b b\\"; countb$
PRINT #1, "define csab cos(a*b)"
PRINT #1, "define csapb cos(a+b)"
PRINT #1, "define snab sin(a*b)"
PRINT #1, "define snapb sin(a+b)"
PRINT #1, "define sa sin(a+0.1)"
PRINT #1, "define ca cos(a+0.1)"
PRINT #1, "define s5 sin(a+0.5)"
PRINT #1, USING "define Vx\ \sa*snapb*ampl*csab+ca*(csapb*ampl*b*←
csab+s5*(ampl*(csab+2.0*snapb)))"; countc$
PRINT #1, USING "define Vy\ \sa*csapb*ampl*csab+ca*(csapb*ampl*b*←
snab+s5*(ampl*(csab+2.0*csapb)))"; countc$
PRINT #1, USING "define Vz\ \ca*csapb*ampl*csab+ca*(snapb*ampl*b*←
snab+s5*(ampl*(snab+2.0*snapb)))"; countc$
                                                                 continued on next page
```

CHAPTER FIVE

```
continued from previous page
PRINT #1,
NEXT x
NEXT y
FOR x = 0 TO 59
   READ clr$
  PRINT #1, "object {"
   PRINT #1, " blob min:"
   FOR y = 0 TO 3
     counta$ = RIGHT$("00" + LTRIM$(STR$(y)), 2)
     countb$ = RIGHT$("00" + LTRIM$(STR$(x)), 2)
     countc$ = counta$ + countb$
     Vx$ = "< Vx" + countc$ + ", "
     Vy$ = "Vy" + countc$ + ",
     Vz$ = " Vz" + countc$ + " >"
     V$ = Vx$ + Vy$ + Vz$
     PRINT #1, USING " str, radius, \
                                                                \,"; V$
   NEXT y
   counta$ = "04"
   countb$ = RIGHT$("00" + LTRIM$(STR$(x)), 2)
  countc$ = counta$ + countb$
  Vx$ = "< Vx" + countc$ + ", "
  Vy$ = " Vy" + countc$ + ", "
  Vz$ = " Vz" + countc$ + " >"
  V$ = Vx$ + Vy$ + Vz$
   PRINT #1, USING " str, radius, \
                                                              \"; V$
   PRINT #1, " u_steps 20"
  PRINT #1, " v_steps 20"
  PRINT #1, " texture {"
  PRINT #1, "
                 rotate <0,0,0>"
  PRINT #1, " }"
  PRINT #1, "}"
  PRINT #1,
NEXT x
CLOSE #1
```

```
kolors:
```

```
DATA "Aquamarine"
DATA "BlueViolet"
DATA "Brown"
DATA "CadetBlue"
```

This program creates SWARM.PI the following Polyray data file, abbreviated here to save space:

```
11
// SWARM.PI
11
// Polyray input file - Jeff Bowermaster
// Based on Eric Deren's MERGE2
start_frame
end_frame
             3500
total_frames 3501
outfile "swrm"
define min 2.0
define str 4.0
define radius 15.0
define u 20
define v 20
// set up the camera
viewpoint {
   from <0,100,-225>
   at <0,0,0>
   up <0,1,0>
   angle 45
   resolution 160,120
   aspect 1.333
// set up background color & lights
background skyblue
light < 220,240,-800>
light < 420,240,-800>
include "\PLY\COLORS.INC"
define ampl 45
define a00 (frame+ 0 )/33.333333
define a01 (frame+ 1 )/33.333333
define a02 (frame+ 2 )/33.333333
define a03 (frame+ 3 )/33.333333
define a04 (frame+ 4 )/33.333333
```

```
continued from previous page
define b00 1.50
define b01 1.53
define b02 1.56
define b03 1.59
define b04 1.62
define b05 1.65
- - -
define b58 3.24
define b59 3.27
define a a00
define b b00
define csab cos(a*b)
define csapb cos(a+b)
define snab sin(a*b)
define snapb sin(a+b)
define ca cos(a+0.1)
define sa sin(a+0.1)
define s5 \sin(a+0.05)
define Vx0000 sa*snapb*ampl*csab+ca*(csapb*ampl*b*csab+s5)*(ampl*←
(csab+2.0*snapb)))
define Vy0000 sa*csapb*ampl*csab+ca*(csapb*ampl*b*snab+s5)*(ampl*←
(csab+2.0*csapb)))
define Vz0000 ca*csapb*ampl*csab+ca*(snapb*ampl*b*snab+s5)*(ampl*←
(snab+2.0*snapb)))
           This last block repeats another 299 times. After all the Vx####, Vy#### and
           Vz#### components are generated, blobs are constructed and unique
           textures applied to each blob:
object {
   blob min:
      str, radius, < Vx0000,
                              Vy0000,
                                        Vz0000 >,
      str, radius, < Vx0100,
                               Vy0100,
                                        Vz0100 >
      str, radius, < Vx0200,
                              V_{y}0200, V_{z}0200 >,
      str, radius, < Vx0300, Vy0300, Vz0300 >,
      str, radius, < Vx0400, Vy0400, Vz0400 >
   u_steps 20
   v steps 20
   texture {
      surface {
        ambient Aquamarine, 0.2
        diffuse Aquamarine, 0.5
        specular white, 0.7
      rotate <0,0,0>
   }
}
```

How It Works

The Quickbasic code SWARM.BAS generates define statements that mimic the swarm simulation code. Positions for all our blob control spheres are generated in the long define listing, and then combined together at the ends to generate the individual blobs:

```
object {
   blob min:
      str, radius, < Vx0000,
                              Vy0000,
                                        Vz0000 >,
      str, radius, < Vx0100,
                              Vy0100,
                                        Vz0100 >,
      str, radius, < Vx0200,
                              Vy0200,
                                        Vz0200 >,
      str, radius, < Vx0300,
                              Vy0300,
                                        Vz0300 >,
      str, radius, < Vx0400, Vy0400,
                                        Vz0400 >
   u_steps 20
   v_steps 20
   texture {
      surface {
        ambient Aquamarine, 0.2
        diffuse Aquamarine, 0.5
        specular white, 0.7
      rotate <0,0,0>
   }
}
```

A sample frame is shown in Figure 5-14.

Comments

The Polyray data file this program creates is huge, but it's a single file that can create as many frames as you want. Here we picked 3,500.

DOS *hates* finding 3,500 files in one directory. As a matter of fact, DOS does not handle directories with more than 256 files in them very well, since

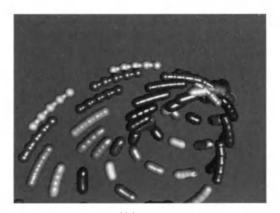


Figure 5-14 Swarming blobs

multiple linked directories are created as well, and writing files requires moving the heads on the hard disk to all these directories as well as to where the file's data is (lots of head movement), and this slows everyone down.

I'd recommend you render 250 targas at a time and write them to their own subdirectories. This is accomplished by changing the heading of the Polyray data file so that the outfile name has the appropriate subdirectory included in the file name. If we wanted the directory "0" to contain the targas for frames 0-249, this would require the header:

and so on. When you're finished, the following directories will contain the following files:

PLY\SWARM\0 contains SWRM000.TGA-SWRM249.TGA PLY\SWARM\250 contains SWRM250.TGA-SWRM499.TGA PLY\SWARM\500 contains SWRM500.TGA-SWRM749.TGA PLY\SWARM\750 contains SWRM750.TGA-SWRM999.TGA PLY\SWARM\1000 contains SWRM1000.TGA-SWRM1249.TGA

Use DTA in list mode to assemble the final flic. Create a file called LIST containing the following lines:

```
0\*.tga
250\*.tga
500\*.tga
750\*.tga
1000\*.tga
```

Then call DTA using

```
DTA alist /ff /oswarm /s3 /c10
```

and you're all set. One fringe benefit here is that by keeping three-numeral and four-numeral targas isolated, you prevent frames 1000-1009 from butting in between frames 100 and 101.

As an alternative to this approach, Dan Farmer pointed out that DTA can deal with LZW archives (it also handles ZIP and ARJ now), so batch processing the targas into a single compressed file is another solution. If Polyray eventually gets internal flic creation support, this could be a moot point, although 3,500 frames, even at 160 x 120, would take well over a week to render on a 486/33.



🏅 5.7 How do I...

Create random numbers in Polyray?

You'll find the code for this in: PLY\CHAPTER5\BROWNIAN

Problem

Polyray doesn't have a scalar random number generator. Scalar numbers are just quantities without location or direction. Many particle systems need to begin with a random distribution of elements or start off with random velocities in random directions. The Brownian function built into Polyray can be used to generate these random numbers for us.

Technique

To illustrate the output of the Brownian function, we construct a unit cube, then ask Brownian to bring into existence a collection of spheres. Shining spotlights through this cube will cast shadows on the walls we've placed for just that purpose, and we can determine what Brownian is actually doing. An example is shown in Figure 5-15.

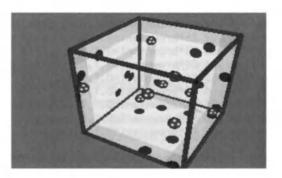


Figure 5-15 Our test rig

Steps

Write the following Polyray data file.

```
BROWNIAN.PI
// Brownian Random Number Generator
11
// By observation, Brownian shifts a vector by
// plus or minus 0.1, so applying Brownian to
// a vector <0.1,0.1,0.1> and then multiplying
// it by 5 gives vector components ranging from
// O to 1, just like RND.
start_frame O
end_frame 25
total_frames 25
outfile brown
include "\PLY\COLORS.INC"
viewpoint {
   from <2.5,2.0,1.5>
   at <0,0,0>
   up <0,1,0>
   angle 35
   resolution 160,100
   aspect 1.43
background SkyBlue
spot_light white, < 5, 0.5, 0.5>, <0.5,0.5>,3,15,25 // x
spot_light white, < 0.5, 5, 0.5>, <0.5,0.5,0.5>,3,15,25 // y
spot_light white, < 0.5, 0.5, 5>, <0.5,0.5>,3,15,25 // z
//object { cylinder < 5, 0.55, 0.55>, <0.5,0.5>,0.01 matte_black }
//object { cylinder < 0.55, 5, 0.55>, <0.5,0.5>,0.01 matte_black }
//object { cylinder < 0.55, 0.55, 5>, <0.5,0.5>,0.01 matte_black }
define v01 [5*brownian(<0.1,0.1,0.1>)]
define v02 [5*brownian(<0.1,0.1,0.1>)]
define v03 [5*brownian(<0.1,0.1,0.1>)]
define v04 [5*brownian(<0.1,0.1,0.1>)]
define v05 [5*brownian(<0.1,0.1,0.1>)]
define v06 [5*brownian(<0.1,0.1,0.1>)]
define v07 [5*brownian(<0.1,0.1,0.1>)]
define v08 [5*brownian(<0.1,0.1,0.1>)]
define v09 [5*brownian(<0.1,0.1,0.1>)]
object {sphere <v01[0][0],v01[0][1],v01[0][2]>,0.05 shiny_red }
object {sphere <v02[0][0],v02[0][1],v02[0][2]>,0.05 shiny red }
object {sphere <v03[0][0],v03[0][1],v03[0][2]>,0.05 shiny_red }
object {sphere <v04[0][0],v04[0][1],v04[0][2]>,0.05 shiny_red }
```

```
object {sphere <v05[0][0],v05[0][1],v05[0][2]>,0.05 shiny_red }
object {sphere <v06[0][0],v06[0][1],v06[0][2]>,0.05 shiny_red }
object {sphere <v07[0][0],v07[0][1],v07[0][2]>,0.05 shiny_red }
object {sphere <v08[0][0],v08[0][1],v08[0][2]>,0.05 shiny_red }
object {sphere <v09[0][0],v09[0][1],v09[0][2]>,0.05 shiny_red }
define a01 < 1, 1, 1>
define a02 < 1, 1, 0>
define a03 < 1, 0, 0>
define a04 < 1, 0, 1>
define a05 <0, 1, 1>
define a06 <0, 1, 0>
define a07 <0, 0, 0>
define a08 <0, 0, 1>
object { cylinder a01,a02,0.02 matte_black }
object { cylinder a02,a03,0.02 matte_black }
object { cylinder a03,a04,0.02 matte_black }
object { cylinder a04,a01,0.02 matte_black }
object { cylinder a05,a06,0.02 matte_black }
object { cylinder a06,a07,0.02 matte_black }
object { cylinder a07,a08,0.02 matte_black }
object { cylinder a08,a05,0.02 matte_black }
object { cylinder a01,a05,0.02 matte_black }
object { cylinder a02,a06,0.02 matte_black }
object { cylinder a03,a07,0.02 matte_black }
object { cylinder a04,a08,0.02 matte black }
object { polygon 4, a05,a06,a07,a08 matte_white}
object { polygon 4, a03,a07,a08,a04 matte white}
object { polygon 4, a02,a06,a07,a03 matte_white}
```

How It Works

The individual components of vectors in Polyray can be extracted if they're initially defined in the form:

```
define v01 [5*brownian(<0.1,0.1,0.1>)]
```

This allows us to call each element as v01[0][0], v01[0][1] and v01[0][2], and use them as scalars. Brownian returns a vector that's within a sphere with a radius of this vector's current length centered on its current position, so Brownian <0.1,0.1,0.1> returns a vector ranging from 0 to 0.2 in each term. Multiplying that value by 5 gives us a random number between 0 and 1.

The animation generates a random collection of spheres that almost look like gas molecules zooming around in a box, although there's no actual motion. They merely appear in random locations.

Note that we could have generated the same animation using object {sphere 5*brownian<0.1,0.1,0.1>,0.05 shiny_red }

but we were interested in determining the values that Brownian returned.



5.8 How do I...

Make a volcano erupt with malted milk balls?

You'll find the code for this in: PLY\CHAPTER5\VOLCANO

Problem

An erupting volcano requires starting with a random distribution of particles moving more or less upward with an initial burst, that under the force of gravity come back to earth and slide down a mountain. This can be handled with the random code (created in Section 5.7) and an enormous data file that tracks the position of a large number of individual elements in the scene.

Technique

A beta version of Polyray has already been written that can generate particle systems animations, so the following method may be considered old fashioned by the time you read this. However, the principles it embodies are valid for any particle systems animation, and can be applied to control any variable you choose.

Steps

Let's write a QuickBasic program that establishes the variables necessary to track the motions of a great many individual particles:

OPEN "c:\ply\dat\volcano\volcano3.pi" FOR OUTPUT AS #1

```
PRINT #1, "start_frame 0"
PRINT #1, "end_frame 200"
PRINT #1, "total_frames 200"
PRINT #1,
PRINT #1, "outfile volc"
PRINT #1,
PRINT #1, "include "; CHR$(34); "\PLY\COLORS.INC"; CHR$(34)
PRINT #1,
PRINT #1, "viewpoint {"
PRINT #1, " from <25,-2,15>"
PRINT #1, " at <0,-2,0>"
```

```
PRINT #1, "
            up <0,1,0>"
PRINT #1, " angle 35"
PRINT #1, "
            resolution 320,200"
PRINT #1, "
              aspect 1.43"
PRINT #1, "
             }"
PRINT #1, "background SkyBlue"
PRINT #1,
PRINT #1, "light 0.6*white, < 10,50, 15>"
PRINT #1, "light 0.6*white, < 0,50, 0>"
PRINT #1,
PRINT #1, "object { cone <0,0,0>,0,<0,-20,0>,20 reflective_white}"
PRINT #1,
PRINT #1, "define rise 1"
PRINT #1, "define run 1"
PRINT #1, "define br 1 // ball radius"
PRINT #1, "define damp 0.5"
PRINT #1, "define g 1"
PRINT #1, "define pi 3.14159"
PRINT #1, "define rad pi / 180"
PRINT #1,
PRINT #1, "
            // set the initial conditions"
FOR item = 0 TO 499
  count$ = RIGHT$("000" + LTRIM$(STR$(item)), 3)
PRINT #1, "if (frame == start_frame) {"
PRINT #1, USING " static define x\ \ O"; count$
PRINT #1, USING " static define y\ \ O"; count$
PRINT #1, USING " static define z\ \ 0"; count$
PRINT #1, USING " define r\ \ [5*brownian(<0.1,0.1,0.1>)]"; count$
PRINT #1, USING " define radius r\ \[0][0]"; count$
PRINT #1, USING " define theta 360 * r\ \[0][1] * rad"; count$
PRINT #1, USING " define phi (30 + 30 * r\ \[0][2]) * rad"; count$
PRINT #1, USING " define velo r\ \[0][0]"; count$
PRINT #1, USING " static define vx\ \ radius * sin(phi) * cos(theta)"; count$
PRINT #1, USING " static define vy\ \ radius * cos(phi) + velo"; count$
PRINT #1, USING "
                   static define vz\ \ radius * sin(phi) * sin(theta)"; count$
PRINT #1, "}"
PRINT #1, "else {"
PRINT #1, USING "
                   static define x\ \ x\ \ + vx\ \"; count$, count$, count$
PRINT #1, USING " static define y\ \ y\ \ + vy\ \"; count$, count$, count$
PRINT #1, USING " static define z\\ z\\ + vz\\"; count$, count$, count$
PRINT #1, "}"
PRINT #1,
PRINT #1, USING "static define vy\ \ vy\ \ - g"; count$, count$
PRINT #1, USING "define dist (x) ^2 +z ^0.5"; count$, count$
PRINT #1, "define ring dist/run"
PRINT #1, USING "if (y\ \ + ring*rise-br < 0)"; count$
PRINT #1, USING " { static define vy\ \ fabs(vy\ \)*damp"; count$, count$
PRINT #1, "}"
PRINT #1, USING "object { sphere <x\ \,y\ \,z\ \>,0.5 shiny__coral }"; count$, \Leftarrow
count$, count$
                                                                continued on next page
```

```
continued from previous page
PRINT #1,
NEXT item
CLOSE #1
```

How It Works

PRINT statements are used to create a canned heading for our Polyray code so that we don't have to deal with it. Then we enter a loop that defines 500 spherical particles, each moving independently. The control code for every particle is identical, but we use the random Brownian function to give each particle its own path. An example of this control code for one particle follows:

```
// set the initial conditions
if (frame == start frame) {
   static define x000 0
   static define y000 0
   static define z000 0
   define r000 [5*brownian(<0.1,0.1,0.1>)]
   define radius r000[0][0]
   define theta 360 * r000[0][1] * rad
   define phi
                 (30 + 30 * r000[0][2]) * rad
   define velo r000[0][0]
   static define vx000 radius * sin(phi) * cos(theta)
   static define vy000 radius * cos(phi) + velo
   static define vz000 radius * sin(phi) * sin(theta)
}
else {
   static define x000 x000 + vx000
   static define y000 y000 + vy000
   static define z000 z000 + vz000
}
static define vy000 vy000 - g
define dist (x000^2 + z000^2)^0.5
define ring dist/run
if (y000 + ring*rise-br < 0)
   { static define vy000 fabs(vy000)*damp
object { sphere \langle x000, y000, z000 \rangle, 0.5 shiny_coral }
```

Each particle is given an initial position, and then a random number between zero and one generated by the Brownian function sets the initial velocity for that particle in polar coordinate notation:

```
x = r*sin(phi)cos(theta)
y = r*cos(phi)
z = r*sin(phi)sin(theta)
```

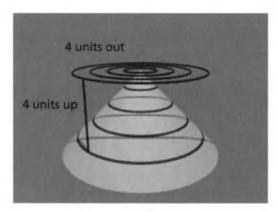


Figure 5-16 A 45° cone, showing the distance vs. height relationship

Use polar coordinates rather than Cartesian notation to effect a spherical eruption. Cartesian coordinates would result in cubical eruptions. Polar coordinates define locations by two angles (phi and theta) and a distance from the origin (x). All particles are given a random upward velocity velo, which adds to the one specified with the polar coordinates. The gravity variable g eventually overcomes this upward motion, and changing g is a convenient means of handling the overall size of the eruption.

After the first frame, the positions of the particles are simply indexed by their velocity. The upward velocity slows under the force of gravity:

static define vy000 vy000 - g

and then we deal with the cone. We defined a white 45° cone for our mountain. The important thing to note about 45° cones is that the distance out from the center (the radius) will always equal the absolute value of the height, provided the tip is placed at the origin, like this one is (Figure 5-16).

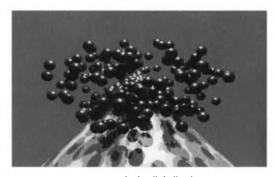
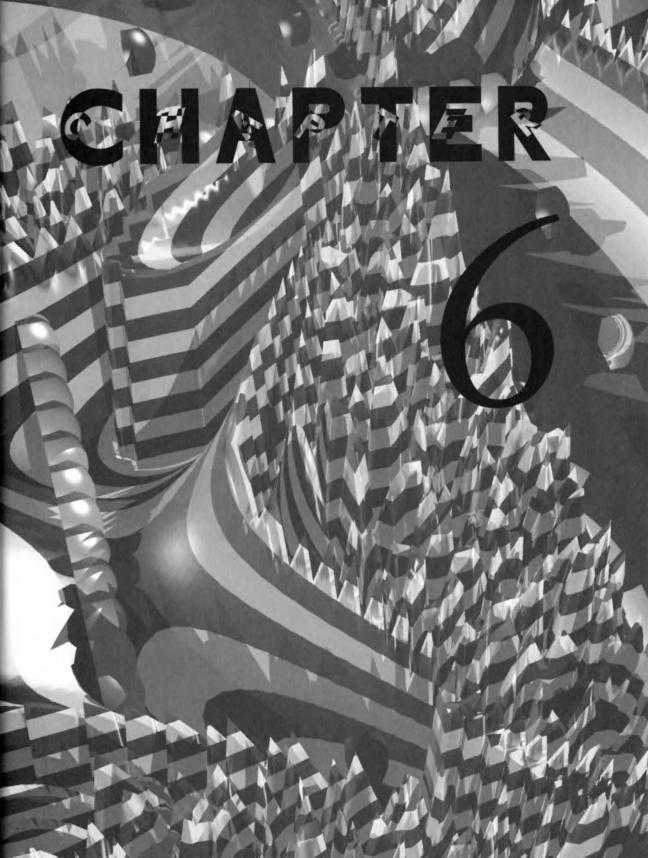


Figure 5-17 Erupting malted milk ball volcano

In the development phase of this animation, a set of rings was used which extend downwards from the eruption point to define a staircase which the balls bounced down. Making those steps finer and finer eventually gave a cone. Whenever the particles y coordinate (which is negative) tries to penetrate this cone, making y000 + ring*rise-br < 0, its velocity flips. In other words, we make it bounce. A frame during the eruption is shown in Figure 5-17.



6

ORCHESTRATION

his chapter focuses on animations which coordinate the movement of collections of objects. We can achieve this control by moving a key element in systems where every object's position in a scene depends on that one element, or write functions keyed to the frame counter and which simultaneously act on various aspects in a scene.



6.1 How do I...

Animate a tumbling buckyball?

You'll find the code for this in: PLY\CHAPTER6\BUCKY

Problem

Moving large numbers of objects collectively in a coordinated fashion is easiest when the entire structure is based on the position of a single key element. Shifting that element dismantles the collection, and returning it to its original position reassembles it. This can allow very complex motions to be generated with something as simple as a single rotation.

Technique

C-60 buckyballs are one member in a whole family of cage carbon compounds. Buckyballs are named after Buckminister Fuller, due to their resemblance to his famous geodesic domes. Their geometry can be derived from two simple rules: pentagons are always surrounded by hexagons, and no three hexagons meet at a single point (see Figure 6-1). Armed with this information, you can go off and build a buckyball model. It's a sequence of translations and rotations, starting with a single pentagon, and it builds successive geometric layers based on the previous ones. The following animation is based on this dependency.

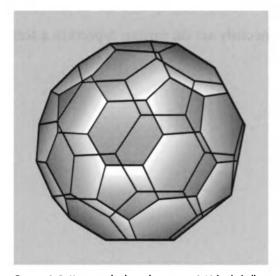


Figure 6-1 Your standard, garden variety C-60 buckyball

The entire buckyball depends on the orientation of the initial pentagon. We'll rotate this member, and all the subsequent members will fly apart into their own little orbits. The modeling job here was not particularly efficient, since as various pentagons and hexagons are folded up, their edges and points overlap the previous layers. In fact, every single vertex gets duplicated, so that this model contains twice the required number of carbon atoms (120 as opposed to 60). However, spheres render quickly and they all become visible as they fly apart.

The background is a chaos map. Chaos describes systems whose future state dramatically depends on initial conditions. Their state cannot be accurately predicted for any length of time. One nice feature chaos maps of this kind have is that they tile perfectly, since they're composed of points mapped on a torus, which is to say the image was developed as a 1 x 1 tile.

Steps

The loops required to create the geometry are not available in Polyray's current syntax, so the following QuickBasic program (BB6PI.BAS) creates individual include files and a batch processor file. The listing is long, so it's been broken up with comments. Note, however, that the program runs as a whole. A rotate subroutine, identical to other ones we've seen, handles the rotations, but has been omitted in this listing.

```
' BB6PI.BAS
' Buckyball Animation Generator - 6
' Jeff Bowermaster, 1993
DECLARE SUB rotate (a,b,c)
Common shared rad, xrotate, yrotate, zrotate
TYPE Vector
  x AS SINGLE
  y AS SINGLE
  z AS SINGLE
END TYPE
DIM penta(5) AS Vector, hex(20, 6) AS Vector
DIM red(16), green(16), blue(16)
' set the screen up with pretty rainbow colors
SCREEN 12
scale = .02
xoff = 320
yoff = 240
WINDOW ((0 - xoff) * scale, (0 - yoff) * scale)-((639 - xoff) * scale, (479 - \Leftarrow
yoff) * scale)
                                                                     continued on next page
```

```
continued from previous page
FOR y = 1 TO 4
  FOR x = 1 TO 4
      colornum = x + ((y - 1) * 4) - 1
      READ red(colornum), green(colornum), blue(colornum)
      KOLOR = 65536 * blue(colornum) + 256 * green(colornum) + red(colornum)
      PALETTE colornum, KOLOR
      COLOR colornum
  NEXT x
NEXT y
'rainbow palette
DATA 0, 0, 0
DATA 32, 0, 0
DATA 42, 0, 0
DATA 58, 16, 0
DATA 63, 32, 0
DATA 58, 56, 0
DATA 16, 42, 0
DATA 0, 30, 36
DATA 0, 20, 40
DATA 0, 10, 48
DATA 0, 0, 63
DATA 20, 0, 53
DATA 23, 0, 29
DATA 19, 7, 17
DATA 50, 40, 45
DATA 63, 63, 63
pi = 3.1415926535#
rad = pi / 180
'= MODIFY THIS LINE SO THE BATCH FILE ENDS UP IN YOUR Polyray Directory =
OPEN "bbanim.bat" FOR OUTPUT AS #1 'batch file to run the
                                  'animations
FOR frame = 0 TO 358 STEP 2
'make a unit pentagram
i = 1
FOR a = frame + 144 TO frame + 432 STEP 72
                                           ' odd phase makes 1 & 2
                                           ' parallel parallel to x-axis
  x = SIN(a * rad) / 1.1755695# ' the 'unit' part
  y = COS(a * rad) / 1.1755695# ' space each member 1 unit apart
  penta(i).x = x
  penta(i).y = y
  penta(i).z = 0
  i = i + 1
NEXT a
```

How It Works

The last section set the palette and defined our first pentagram. The next section will take each side of this pentagram and attach a hexagram to it so that both shapes share two vertices. Actually only two points of the pentagram are used, as they're positioned conveniently with respect to the origin to exploit the symmetry.

We use the nomenclature

hex(a,b), where a represents a particular hexagram, and b represents a particular sphere within that hexagram. So hex (1,2) is the second sphere in the first hexagram.

Start with the b=2 sphere, translate it to the origin, dragging the b=1 sphere along with it, rotate b=1 240° about b=2, then translate b=2 (+baggage) back to its original position. Repeat with b=3, rotating b=2, (then 4,3 and 5,4) leapfrogging a hex into shape:

```
xrotate = 0
 yrotate = 0
  zrotate = 240
' make hex elements 1 & 2 two sides vertices of the pentagram
 FOR b = 1 TO 2
   hex(1, b) = penta(b)
 NEXT b
 a = 1
 FOR b = 3 TO 6
'translate
   hex(a, b).x = hex(a, b - 2).x - hex(a, b - 1).x
   hex(a, b).y = hex(a, b - 2).y - hex(a, b - 1).y
   hex(a, b).z = hex(a, b - 2).z - hex(a, b - 1).z
'rotate
CALL rotate (hex(a,b).x, hex(a,b).y, hex(a,b).z)
'translate back
   hex(a, b).x = hex(a, b).x + hex(a, b - 1).x
   hex(a, b).y = hex(a, b).y + hex(a, b - 1).y
   hex(a, b).z = hex(a, b).z + hex(a, b - 1).z
 NEXT b
```

Now we have the first hexagram. What we'll do next is rotate this shape four times to construct the others, but while we're here, let's also do the x-axis

rotation to form a bowl. The hex's 1-2 points are parallel with the x axis (but below it). Translate them up onto the x axis (adding their y offsets), rotate the whole hex by 37.3775° about x, then translate it back down. By the way, 37.3775° was determined experimentally. It was determined by having two hexagrams attached to the pentagram and manually varying the x rotation until the third element on one coincided with the sixth element on the other. The temporary code used in determining this has been commented out, but it shows you how it was done:

```
' translate
' save the y offset
 yoffset = hex(a, 1).y
 FOR b = 1 TO 6
   hex(a, b).y = hex(a, b).y - yoffset
' rotate
   xrotate = 37.3775
   yrotate = 0
   zrotate = 0
   CALL rotate (hex(a,b).x, hex(a,b).y, hex(a,b).z)
'translate back
   hex(a, b).y = hex(a, b).y + yoffset
   'CIRCLE (hex(a, b).x, hex(a, b).y), .1, 3
 NEXT b
 xrotate = 0
' rotate this one 72° four times
' to generate the others
 FOR a = 2 TO 5
   zrotate = 72 * (a - 1)
   FOR b = 1 TO 6
     tempx = hex(1, b).x
     tempy = hex(1, b).y
     tempz = hex(1, b).z
     CALL rotate (tempx, tempy, tempz)
     hex(a, b).x = tempx
     hex(a, b).y = tempy
     hex(a, b).z = tempz
    ' CIRCLE (hex(a, b).x, hex(a, b).y), .1, b + 2
```

We're now going to rotate one of the hexagrams into the *x-y* plane, center it on the *x* axis, and then rotate hexes 3 and 4 by 120°, which moves them conveniently into the next level. Since the current center of the buckyball isn't at the origin, rotation into the *x-y* plane will place the center of the hex some distance down in *y*. Rotating hexes 3 and 4 about *x* and translating its center moves it to 0,0, *whatever*, where *whatever* doesn't matter at this point:

```
FOR a = 3 TO 4
 xrotate = -37.3775
 yrotate = 0
 zrotate = 0
 FOR b = 1 TO 6
    tempx = hex(a, b).x
    tempy = hex(a, b).y
    tempz = hex(a, b).z
CALL rotate (tempx, tempy, tempz)
    hex(a + 3, b).x = tempx
    hex(a + 3, b).y = tempy + 1.4129046# ' another determined factor
    hex(a + 3, b).z = tempz
                                        ' sorry about that
 NEXT b
' now rotate by 120° in z
  xrotate = 0
 yrotate = 0
  zrotate = 120
  FOR b = 1 TO 6
    CALL rotate (hex(a + 3,b).x, hex(a + 3,b).y, hex(a + 3,b).z)
 NEXT b
' rotate and translate back to the original position
```

continued on next page

```
continued from previous page
  xrotate = 37.3775
  yrotate = 0
  zrotate = 0
  FOR b = 1 TO 6
    CALL rotate (hex(a +3.b).x, hex(a +3.b).y - 1.4129046, hex(a +3.b).z)
    'CIRCLE (hex(a + 3, b).x, hex(a + 3, b).y), .1, a + 3
  NEXT b
NEXT a
' take 6 and 7 and rotate 72ø about z four more times
xrotate = 0
yrotate = 0
FOR z = 1 TO 4
FOR a = 6 TO 7
  FOR b = 1 TO 6
    zrotate = z * 72
    tempx = hex(a, b).x
    tempy = hex(a, b).y
    tempz = hex(a, b).z
    CALL rotate (tempx, tempy, tempz)
    hex(a + 2 * z, b).x = tempx
    hex(a + 2 * z, b).y = tempy
    hex(a + 2 * z, b).z = tempz
    'CIRCLE (hex(a + 2 * z, b).x, hex(a + 2 * z, b).y), .1, 3 + b
  NEXT b
NEXT a
NEXT z
```

OK, now we cheat big time. It turns out the final piece is an inside-out copy of the opposite side, only rotated by 36° . Since we need the hexes anyway, simply copy hexagrams 1 through 5, reflect in the x-y plane at z=0, rotate 36° and translate them -4.67 in z. This factor -4.67 was again determined by manual fitting:

```
xrotate = 0
yrotate = 0
zrotate = 36

FOR a = 1 TO 5
  FOR b = 1 TO 6

  tempx = hex(a, b).x
  tempy = hex(a, b).y
  tempz = hex(a, b).z
  CALL rotate (tempx,tempy,tempz)
```

```
hex(a + 15, b).x = tempx
    hex(a + 15, b).y = tempy
    hex(a + 15, b).z = ABS(tempz) - 4.65491 ' lucked out
    'CIRCLE (hex(a + 15, b).x, hex(a + 15, b).y), .1, b + 3
  NEXT b
NEXT a
' for a dynamite visual effect, tumble the whole mess in x
xrotate = frame
yrotate = 0
zrotate = 0
FOR a = 1 TO 20
  FOR b = 1 TO 6
    tempx = hex(a, b).x
    tempy = hex(a, b).y
    tempz = hex(a, b).z + 2.3274455# ' off center adjust
    CALL rotate (tempx, tempy, tempz)
    hex(a, b).x = tempx
    hex(a, b).y = tempy
    hex(a, b).z = tempz
    'CIRCLE (hex(a + 15, b).x, hex(a + 15, b).y), .1, b + 3
  NEXT b
NEXT a
```

Now let's write the file and a preview looking down the z axis (Figure 6-2). You'll probably first want to create a subdirectory (e.g., BUCK to hold the .PIs and TGAs). This step will take 2MB.

```
pref$ = "bc"
  frame.count$ = RIGHT$("000" + LTRIM$(STR$(frame)), 3)
  frame.dat$ = pref$ + frame.count$ + ".inc"
  frame.tga$ = pref$ + frame.count$ + ".tga"
 OPEN frame.dat$ FOR OUTPUT AS #2
  PRINT #1, "echo include "; CHR$(34); frame.dat$; CHR$(34); " > bbs.inc"
  PRINT #1, "echo define frm "; frame; " >> bbs.inc"
 PRINT #1, "\ply\polyray bucky.pi -o "; frame.tga$
 CLS
  FOR a = 1 TO 20
     FOR b = 1 TO 6
        CIRCLE (hex(a, b).x, hex(a, b).y), .35, INT(1.5 * (hex(a, b).z + 5))
        PRINT #2, USING "object { sphere < ##.###,##.#### >,0.35 \(\infty\)
bumpy__coral}"; hex(a, b).x; hex(a, b).y; hex(a, b).z
        NEXT b
     NEXT a
                                                                  continued on next page
```

CHAPTER SIX

continued from previous page
CLOSE #2
NEXT frame

CLOSE #1

Two types of files are generated. There will be 180 BCxxx.INC files that define the positions of the 120 spheres comprising the buckyball, and a BBANIM.BAT file to call the master program, BUCKY.PI. BBANIM writes a file telling BUCKY.PI which geometries to include and what frame number it's on, which controls the sliding image map in the background.

BBANIM.BAT looks like this:

```
echo include "bc000.inc" > bbs.inc
echo define frm 0 >> bbs.inc
\ply\polyray bucky.pi -o bc000.tga
echo include "bc002.inc" > bbs.inc
echo define frm 2 >> bbs.inc
\ply\polyray bucky.pi -o bc002.tga
echo include "bc004.inc" > bbs.inc
echo define frm 4 >> bbs.inc
\ply\polyray bucky.pi - o bc004.tga
```

BBS.INC for frame 0 looks like this:

include "bc000.inc"
define frm 0

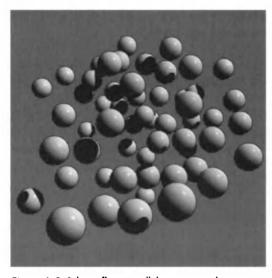


Figure 6-2 Spheres fly out in all directions; pandemonium reigns

And the master program BUCKY.PI looks like this:

```
// BUCKY.PI
// Polyray file
// Tumbling Buckyball by Jeff Bowermaster
include "\PLY\COLORS.INC"
viewpoint{
   from < 0.0, 1.0, -9.0 >
   at < 0.0, -0.45, 0.0 >
      < 0.0, 1.0, 0.0 >
   angle 50
   resolution 320,200
   aspect 1.433
}
define bumpy_coral
texture {
   special surface {
      color <1.5,0.75,0>
      normal N + (dnoise(3*W) - white/2)
      ambient 0.2
      diffuse 0.3
      specular white, 0.7
      microfacet Cook 5
   scale <0.05, 0.05, 0.05>
light white ^{2}, ^{0.0}, ^{8.0}, ^{-4.0}
include "bbs.inc"
define the_image image("fire.tga")
define flames
   texture {
      special surface {
         color planar_imagemap(the_image, P, 1)
         ambient 0.2
         diffuse 0.8
      rotate <90,0,0>
      scale <20,20,20>
      translate <-0.5, frm/9, -0.5>
   }
object { disc <0,0,10>,<0,0,1>, 20 flames }
```

This is a fairly standard example of external animation using Polyray. A series of complex geometric forms are generated using QuickBasic code, and written to individual include files. A hard-built master file is created that contains the static information, things like the camera location, lighting, and

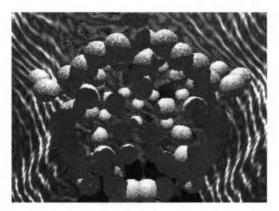


Figure 6-3 Exploding buckyball

textures, as well as variables we may need to change from frame to frame. A line in this master file references a single include file. It contains three things: 1) the name of one of the 180 geometry files the QuickBasic program created, 2) a Frame number, and 3) a line to start Polyray rendering. We get a DOS batch file (BBANIM.BAT) to change the information in this include file for us from frame to frame. This is handled using the redirection operators (> and >>), where screen output is rerouted to a file. A sample image is shown in Figure 6-3.

Comments

It's easy to forget that animations running in batch mode have no frame counter, since in each instance the ray tracer is only rendering a single file. This means that if elements like the chaos background need to be put under the control of the frame counter, we need to include this changing information in the batch file. Also, the names of the image files have to be contained in this file, because we no longer have automatic sequential output.



6.2 How do I...

Tumble a group of interlocking rings?

You'll find the code for this in: PLY\CHAPTER6\RINGS

Problem

Interlocking polygons constitute an interesting set of symmetrical figures. We map the vertices onto the surface of a sphere and move them around so all the points remain on the sphere, and while we can't do it with this code, it's

almost possible to generate a set in which all the vertices are spaced equally from one another around a sphere. A whole series of interlocking rings can be created programmatically and tumbled to produce a sloshing, strangely soothing animation.

Technique

Karl Weller made several great examples of interlocking rings. If you stare at them for a few days, you can derive the following QuickBasic code that generates interlocking polygons containing any number of vertices. The ones with three and five vertices seem to look the best, while the ones with an even numbers of vertices generate rings that physically overlap. It's very pleasant to place these figures into an interlocking motion and watch them intertwine.

The following listing (RINGER.BAS) is the simulation code for the interlocking rings. Feel free to change the *sides* variable and see the types of interlocking rings this code creates:

```
DECLARE SUB rotate (a, b, c)
COMMON SHARED xrotate, yrotate, zrotate, rad
' RINGER.BAS - Interlocking Ring Creation Program
' (c) 1993, Jeff Bowermaster, Splat! Graphics
' creates interlocking tumbling polygons
' based on Geometry supplied by Karl Weller in his Ring images
' change the variable sides from 3-9 for various shapes
' three and five seem to work the best, even numbers produce degenerate
' (overlapping) polygons
TYPE Vector
  x AS SINGLE
  y AS SINGLE
  z AS SINGLE
END TYPE
DIM p(10, 9) AS Vector, po(10, 9) AS Vector, pf(10) AS Vector
pi = 3.1415923#
rad = pi / 180
' change this from 3 to 9 for interesting effects
sides = 5
SCREEN 12
WINDOW (-1.6, -1.2)-(1.6, 1.2)
DO WHILE INKEY$ = ""
k = k + 1
```

```
continued from previous page
' make the first Polygon
FOR b = 1 TO sides
   ang = b * 360 / sides
   pf(b).x = COS(ang * rad)
   pf(b).y = SIN(ang * rad)
   pf(b).z = 0
   ' make a copy at more or less a random angle
   xrotate = 75 * SIN(k * rad)
   yrotate = 75 * COS(2 * k * rad)
   zrotate = k
  'for a five-sided static interlocking figure, uncomment these lines
  'xrotate = 63.434948
  'yrotate = 0
  'zrotate = 0
   CALL rotate(pf(b).x, pf(b).y, pf(b).z)
   p(1, b).x = pf(b).x
   p(1, b).y = pf(b).y
   p(1, b).z = pf(b).z
NEXT b
' make all the other ones
FOR a = 2 TO sides + 1
  FOR b = 1 TO sides
      p(a, b).x = pf(b).x
      p(a, b).y = pf(b).y
      p(a, b).z = pf(b).z
      xrotate = 0
      yrotate = 0
      zrotate = 360 / (2 * sides) + a * 360 / sides
      CALL rotate(p(a, b).x, p(a, b).y, p(a, b).z)
   NEXT b
NEXT a
' The rest is just draw-undraw code for the motions
FOR a = 1 TO sides + 1
   FOR b = 1 TO sides
      CIRCLE (po(a, b).x, po(a, b).y), .1, 0
   NEXT b
NEXT a
```

```
FOR a = 1 TO sides + 1
   FOR b = 1 TO sides -1
      LINE (po(a, b).x, po(a, b).y)-(po(a, b + 1).x, po(a, b + 1).y), 0
   NEXT b
   LINE (po(a, sides).x, po(a, sides).y)-(po(a, 1).x, po(a, 1).y), 0
NEXT a
FOR a = 1 TO sides + 1
   FOR b = 1 TO sides
      po(a, b).x = p(a, b).x
      po(a, b).y = p(a, b).y
      po(a, b).z = p(a, b).z
   NEXT b
NEXT a
FOR a = 1 TO sides + 1
   FOR b = 1 TO sides
      CIRCLE (p(a, b).x, p(a, b).y), .1, a
   NEXT b
NEXT a
FOR a = 1 TO sides + 1
   FOR b = 1 TO sides -1
      LINE (p(a, b).x, p(a, b).y)-(p(a, b + 1).x, p(a, b + 1).y), a
   NEXT b
   LINE (p(a, sides).x, p(a, sides).y)-(p(a, 1).x, p(a, 1).y), a
NEXT a
L00P
SUB rotate (x, y, z)
       x0 = x
       y0 = y
       z0 = z
       y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
       z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)
       x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
       y2 = y1
       z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)
       x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)
       y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)
       z3 = z2
       x = x3
       y = y3
       z = z3
END SUB
```

How It Works

A foundation polygon pf(t) with the selected value of *sides* is generated, and then a copy is generated and rotated into another location using tumbling code that it rocks back and forth on the x and y axes $\pm 75^\circ$, with a constant z rotation:

```
xrotate = 75 * SIN(k * rad)
yrotate = 75 * COS(2 * k * rad)
zrotate = k
```

This gives us a random starting element. A second series of rotations makes the required number of copies to generate our final interlocking ring structure:

```
zrotate = 360 / (2 * sides) + a * 360 / sides
```

While there's really only one correct angle for Karl Weller's original model (a pentagonal structure requires 63.434948°), this code rocks the figure back and forth into a myriad of other configurations. The rest of the code above deals with drawing and undrawing the screen. Figure 6-4 shows an example of five membered rings as they tumble.

The textures for this animation were selected from the default colors in Polyray. We need six colors that won't do evil things to each other, and so we settled on earthy golds and tans.

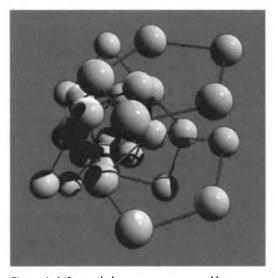


Figure 6-4 Six interlocking pentagons in a tumbling configuration

Polyray Code

microfacet Phong 5

The following code makes an interlocking polygon model inside Polyray.

```
// RINGER.PI - Six Interlocking Pentagons Jostle in Front of Corner Mirror
//
       Splat! Graphics
//
       Based on geometry supplied by Karl Weller
start_frame 0
end_frame 359
total_frames 360
outfile "rng"
define a 1
define pi 3.1415927
define rad pi / 180
viewpoint {
   from <3,3,3>
   at <0,0,0>
   up <0,1,0>
   angle 30
   resolution 320,200
   aspect 1.433
background MidnightBlue
spot_light <0.75,0.75,0.75>, <0,30,0>, <0,0,0>, 3, 5, 20
light <0.6,0.3,0.15>,<0,0,0>
// six default colors set as textures
define txr007
texture {
   surface {
      ambient Coral, 0.2
      diffuse Coral, 0.8
      specular white, 0.6
      microfacet Phong 5
      reflection white, 0.25
      transmission white, 0, 0
   }
define txr070
texture {
   surface {
      ambient Yellow, 0.2
      diffuse Yellow, 0.8
      specular white, 0.6
```

continued on next page

CHAPTER SIX

```
continued from previous page
      reflection white, 0.25
      transmission white, 0, 0
      }
   }
define mirror
texture {
   surface {
      ambient white, 0.1
      diffuse white, 0.2
      specular O
     reflection white, 1
   }
define sides 5
define ang 1 * 360 / sides
define x1 cos(ang * rad)
define y1 sin(ang * rad)
define z1 0
define ang 2 * 360 / sides
define x2 cos(ang * rad)
define y2 sin(ang * rad)
define z2 0
define ang 3 * 360 / sides
define x3 cos(ang * rad)
define y3 sin(ang * rad)
define z3 0
define ang 4 * 360 / sides
define x4 cos(ang * rad)
define y4 sin(ang * rad)
define z4 0
define ang 5 * 360 / sides
define x5 cos(ang * rad)
define y5 sin(ang * rad)
define z5 0
define thk 0.10
// sloshing the figure about, 75 (rather than 90) degrees keeps it from
// ever developing somewhat boring five-way symmetry
define xr 75 * sin(frame*rad)
define yr 75 * cos(2 * frame * rad)
define zr frame
define ring
  object {
```

```
object { sphere <x1, y1, z1>, thk }
    + object { cone <x1, y1, z1>, thk, <x2, y2, z2>, thk }
    + object { sphere <x2, y2, z2>, thk }
    + object { cone <x2, y2, z2>, thk, <x3, y3, z3>, thk }
    + object { sphere \langle x3, y3, z3 \rangle, thk }
   + object { cone <x3, y3, z3>, thk, <x4, y4, z4>, thk }
    + object { sphere <x4, y4, z4>, thk }
    + object { cone <x4, y4, z4>, thk, <x5, y5, z5>, thk }
    + object { sphere <x5, y5, z5>, thk }
   + object { cone <x5, y5, z5>, thk, <x1, y1, z1>, thk }
      rotate <xr,yr,zr>
   }
object { ring rotate <-63.434948,0,0> txr007 }
object { ring rotate <0, 0, 36+0*72> txr021 }
object { ring rotate <0, 0, 36+1*72> txr022 }
object { ring rotate <0, 0, 36+2*72> txr039 }
object { ring rotate <0, 0, 36+3*72> txr063 }
object { ring rotate <0, 0, 36+4*72> txr070 }
// corner mirror
object { disc <-2,0,0>,<1,0,0>,8 mirror}
object { disc <0,-2,0>,<0,1,0>,8 mirror}
object { disc <0,0,-2>,<0,0,1>,8 mirror}
```

The six interlocking pentagonal rings are defined inside Polyray. A single ring is defined first, which is then rotated into the five other locations. We slosh the figure about by 75° (rather than 90), to keep it from ever developing somewhat boring five-way symmetry. We place the figure in a corner mirror (three perpendicular mirrors) that repeat the geometry off into the distance. A sample image is shown in Figure 6-5.

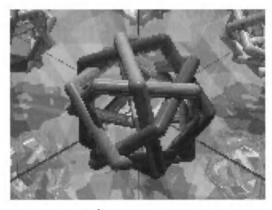


Figure 6-5 Interlocking rings

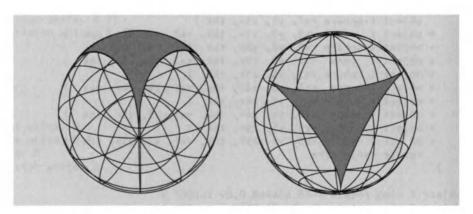


Figure 6-6 A deltoid mapped onto a sphere provides a shifting viewpoint

Comments

A minor variation of this animation involves moving the viewpoint around in a triangular shape called a deltoid mapped onto a sphere (see Figure 6-6). It's controlled by the following viewpoint code:

```
define vx 2 * a * cos(frame * rad) + a * cos(2 * frame * rad) define vy 2 * a * sin(frame * rad) - a * sin(2 * frame * rad) define vz (27 - ((vx) ^ 2 + (vy) ^ 2)) ^ 0.5
```

Since the ringed object is inside a corner cube, the changing viewpoint highlights the multiple reflections and gives the figure a terrific sense of space, as shown in RINGZ.PI.

```
// RINGZ.PI - Six Interlocking Pentagons Jostle in Front of Corner Mirror
11
              Viewed from Different Angles
11
      1993, Jeff Bowermaster,
//
       Splat! Graphics
11
       Polyray v1.5 data file
11
       Based on geometry supplied by Karl Weller
start_frame O
end_frame 359
total_frames 360
outfile "ring"
define a 1
define pi 3.1415927
define rad pi / 180
// deltoid path - looks like a triangle sucking in it's cheeks
// the vz term keeps it tethered to the origin
define vx 2 * a * cos(frame * rad) + a * cos(2 * frame * rad)
```

```
define vy 2 * a * sin(frame * rad) - a * sin(2 * frame * rad)
define vz (27 - ((vx) ^ 2 + (vy) ^ 2)) ^ 0.5

define view rotate (<vx,vy,vz>,<-45,0,-45>)

viewpoint {
    from view
    at <0,0,0>
    up <0,1,0>
    angle 30
    resolution 320,200
    aspect 1.433
    }
...
...
```

The remainder of the data file is the same as RINGER.PI. The deltoid viewpoint is defined as a constant distance from the z axis using the equation for a circle, which gives us a triangle mapped onto a sphere. This path is rotated 45° about x and z to give a more diagonal vantage point to our corner cube.



6.3 How do I...

View color space?

You'll find the code for this in: PLY\CHAPTER6\SPHERE

Problem

There are several ways of viewing color space. Color has three components, just like Cartesian space, so you can think of locations in space as corresponding to particular colors. It's not surprising that cubes are frequently used to display the three primary colors (red, green, and blue) down each coordinate axis. You run each color down one axis of the cube, ending up with a cube with black, red, green, blue, magenta, cyan, yellow, and white at its corners, and all the other colors in between. Unfortunately, a solid cube hides most of the colors inside it, and we have to tumble it to show all the ones on the outside. We need to break it up somehow to see more colors.

On a sphere, phasing green and blue 90° apart around the equator and running red linearly from top to bottom creates an interesting variation. This is what Karl Weller did, and we'll use his color model to generate the following tumbling/rotating collection of spheres painted with every color in the rainbow and then some. We'll want to move it so that we get to see all the colors.

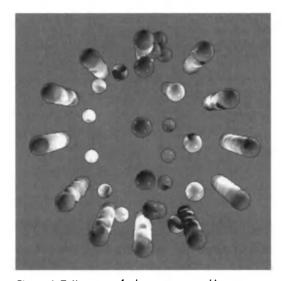


Figure 6-7 Nine rings of spheres prior to tumbling

Technique

What we make here is nine rings, seen edge on in Figure 6-7, each comprised of 16 spheres. We'll assign each sphere its own color, with the amounts of blue and green based on angle around the ring, and red based on which ring it's on. In order to actually see all the colors, we'll make each ring rotate at a different speed. The upper four rings will move clockwise, the lower four rings will move counterclockwise, and the middle ring will slosh back and forth like a washing machine agitator. For a final effect, we'll tumble the twisting structure end over end, and this will bring every sphere into view at some point during the animation.

Steps

The following code (SPHE.BAS) automtically writes the entire Polyray file:

```
' SPHE.BAS
SCREEN 12
WINDOW (-320, -240)-(320, 240)

OPEN "sphe.pi" FOR OUTPUT AS #1

PRINT #1, "//"
PRINT #1, "// SPHERE.PI"
PRINT #1, "//"
PRINT #1, "//"
PRINT #1, "// Polyray input file - Jeff Bowermaster"
PRINT #1, "// Thanks to Karl Weller for the Colored Space Definition"
```

```
PRINT #1,
PRINT #1, "// define the range of the animation"
PRINT #1,
PRINT #1, "start_frame
PRINT #1, "end_frame
                        719"
PRINT #1, "total_frames 720"
PRINT #1, "outfile "; CHR$(34); "sphe"; CHR$(34)
PRINT #1,
PRINT #1, "define pi 3.1415927"
PRINT #1, "define rad pi/180"
PRINT #1,
PRINT #1, "// set up the camera"
PRINT #1, "viewpoint {"
PRINT #1, " from <90,40,400>"
PRINT #1, "
             at <0,0,5>"
PRINT #1, "
            up <0,1,0>"
PRINT #1, " angle 45"
PRINT #1, "
             resolution 320,200"
PRINT #1, "
             aspect 1.433"
PRINT #1, "
             }"
PRINT #1,
PRINT #1, "// set up background color & lights"
PRINT #1, "background MidnightBlue *0.5"
PRINT #1, "light <0.75, 0.75, 0.75>, < 0,0,0>"
PRINT #1,
pi = 3.1415927#
rad = pi / 180
r = 140
ring = 1
FOR i = -80 TO 80 STEP 20 ' Nine Rings
   y = SIN(i * rad) * r
   r2 = COS(i * rad) * r
   sphr = 6 * (COS(i * rad) + 1) * (COS(i * rad) + 1)
   ring$ = "ring" + LTRIM$(STR$(ring))
   PRINT #1, USING "define \ \ object {"; ring$
   ring = ring + 1
   FOR j = 0 TO 337.5 STEP 22.5
      x = SIN(j * rad) * r2
      z = COS(j * rad) * r2
      red = ((y / r) + 1!) / 2!
      green = ((x / r2) + 1!) / 2!
      blue = ((z / r2) + 1!) / 2!
      IF j = 0 THEN
         PRINT #1, USING " object { sphere <###.##, ####.##, ####.##>, ←
###.## texture { surface { color <#.###, #.###, #.### > ambient 0.4 diffuse ←
0.6 reflection 0.95 microfacet Phong 5 }}}"; x, y, z, sphr, red, green, blue
      ELSE
         PRINT #1, USING " + object { sphere <###.##, ####.##, ####.##>, \(\infty\)
###.## texture { surface { color <#.###, #.###, #.### > ambient 0.4 diffuse ←
0.6 reflection 0.95 microfacet Phong 5 }}}"; x, y, z, sphr, red, green, blue
      END IF
                                                                 continued on next page
```

```
continued from previous page
      CIRCLE (x, y), 10
   NEXT j
   PRINT #1, "}"
   PRINT #1,
NEXT i
FOR ring = 1 \text{ TO } 9
   ring$ = "ring" + LTRIM$(STR$(ring))
   IF ring < 5 THEN
      PRINT #1, USING "object { \ \ rotate <0.0, frame * #.#, 0.0> }"; \Leftarrow
ring$, ring / 2
   ELSE
      IF ring = 5 THEN
          PRINT #1,
         PRINT #1, USING "object { \ \ rotate <0.0, 180*sin(frame * rad), \Leftarrow
0.0> }"; ring$
          PRINT #1,
      ELSE
          PRINT #1, USING "object { \ \ rotate <0.0, frame * ##.#, 0.0> }"; \leftarrow
ring\$, -(10 - ring) / 2
      END IF
   END IF
NEXT ring
CLOSE #1
```

How It Works

Most of the file SPHERE.PI is color definition and sphere placement. There are nine rings containing 16 spheres each. These rings rotate, with the smaller rings rotating the slowest. The equator ring sloshes back and forth. The entire model flips end over end, and you get to see almost every conceivable color a ray tracer can generate.

```
//
// SPHERE.PI
//
// Polyray input file - Jeff Bowermaster
// Thanks to Karl Weller for the Colored Space Definition
// define the range of the animation
start_frame 0
end_frame 359
total_frames 360
outfile "sphe"

define pi 3.1415927
define rad pi/180
define r 140.0
// set up the camera
viewpoint {
```

```
from <0,0,-400>
   at <0,0,0>
   up <0,-1,0>
   angle 50
   resolution 320,200
   aspect 1.433
   }
// set up background color & lights
background MidnightBlue*0.75
light <0.75,0.75,0.75>,<0,0,0>
define ring1 object {
     object { sphere <
                          0.00, -137.87, 24.31>, 8.26 texture { surface { \Leftarrow
color <0.008, 0.500, 1.000 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere <
                         9.30, -137.87, 22.46>, 8.26 texture { surface { ←
color <0.008, 0.691, 0.962 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere < 17.19, -137.87, 17.19>,
                                                     8.26 texture { surface { ←
color <0.008, 0.854, 0.854 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere < 22.46, -137.87, 9.30>, 8.26 texture { surface { ←
color <0.008, 0.962, 0.691 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere < 24.31, -137.87, -0.00>, 8.26 texture { surface { ←
color <0.008, 1.000, 0.500 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere < 22.46, -137.87, -9.30>, 8.26 texture { surface { \Leftarrow
color <0.008, 0.962, 0.309 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere < 17.19, -137.87, -17.19>, 8.26 texture { surface { ←
color <0.008, 0.854, 0.146 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere <
                         9.30, -137.87, -22.46>, 8.26 texture { surface { ←
color <0.008, 0.691, 0.038 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere < -0.00, -137.87, -24.31>, 8.26 texture { surface { \Leftarrow
color <0.008, 0.500, 0.000 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere < -9.30, -137.87, -22.46>, 8.26 texture { surface { ←
color <0.008, 0.309, 0.038 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere < -17.19, -137.87, -17.19>, 8.26 texture { surface { ←
color <0.008, 0.146, 0.146 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere < -22.46, -137.87, -9.30>, 8.26 texture { surface { \Leftarrow
color <0.008, 0.038, 0.309 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere < -24.31, -137.87, 0.00>,
                                                     8.26 texture { surface { ←
color <0.008, 0.000, 0.500 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
                                                                    continued on next page
```

```
continued from previous page
   + object { sphere < -22.46, -137.87, 9.30>, 8.26 texture { surface { \Leftarrow
color <0.008, 0.038, 0.691 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere < -17.19, -137.87, 17.19>, 8.26 texture { surface { \Leftarrow
color <0.008, 0.146, 0.854 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
   + object { sphere < -9.30, -137.87, 22.46>, 8.26 texture { surface { \Leftarrow
color <0.008, 0.309, 0.962 > ambient 0.4 diffuse 0.6 reflection 0.95 \Leftarrow
microfacet Phong 5 }}}
(continues like this for 8 more rings)
define mass
   object {
      object { ring1 rotate <0.0, frame * 1.0, 0.0> }
    + object { ring2 rotate <0.0, frame * 2.0, 0.0> }
    + object { ring3 rotate <0.0, frame * 3.0, 0.0> }
    + object { ring4 rotate <0.0, frame * 4.0, 0.0> }
    + object { ring5 rotate <0.0, 180*sin(frame * rad), 0.0> }
    + object { ring6 rotate <0.0, frame * -4.0, 0.0> }
    + object { ring7 rotate <0.0, frame * -3.0, 0.0> }
    + object { ring8 rotate <0.0, frame * -2.0, 0.0> }
    + object { ring9 rotate <0.0, frame * -1.0, 0.0> }
mass { rotate <frame,0,0> }
```

The Quickbasic code takes care of specifying the colors and positions for all our spheres, surrounding one large sphere. The final definition for *mass* calls all these rings together and sets them rotating about the *y* axis. When we call *mass*, we flip it end over end about the *x* axis. It's shown in Figure 6-8.

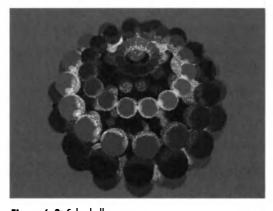
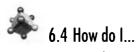


Figure 6-8 Color ball



Do something original with the MTV logo?

You'll find the code for this in: PLY\CHAPTER6\MTV

Problem

MTV airs an amazing variety of creative modifications to its basic logo. They leap, they flame, they form out of lawns, they're tattoos, they're puppets, they're violent, twisted, fluorescent, fluid; a half hour of nothing but MTV logos set to music would be pretty entertaining. It's a real challenge to come up with something slick that hasn't already been tried.

Technique

This animation generates an MTV logo bobbing above a rippling yellow sea, and spirals a bunch of throbbing spheres around it that alternate between mirror and glass every other frame. The basic idea was kicked around on the You Can Call Me Ray BBS on Christmas 1991 by John Hammerton, David Mason, Douglas Otwell, and myself in both POV-Ray (or DKB-Trace) and the then brand-new Polyray ray tracer. While the POV-Ray program required spiral include files and batch processing, Polyray can do it all using a single data file and internal animation support.

We break the work down into three parts: generate the logo, create the spiral, and merge the two elements together for the final animation.

Steps

The logo part is simple but somewhat tedious. Tape MTV, and snag one of their logos. Slap a baggie or plastic wrap up on your TV screen, the static electricity will hold it in place, bring up the logo freeze frame and trace it using a grease pencil or plastic marker. Slap the tracing on your computer screen, and use whatever CAD or Paint program you have that will give you x,y coordinates for your mouse, and generate the coordinates for the features in the logo. If you don't have access to such programs, you could just use graph paper, but the TV part is much simpler to generate when you can grow spheres interactively.

The M and TV parts were created at different times, which explains why in the following code there's some minor scaling going on. We create the M using four boxes, two of them rotated, and use a combination of spherical and cylindrical blobs for the TV part. Believe it or not, all of this stuff

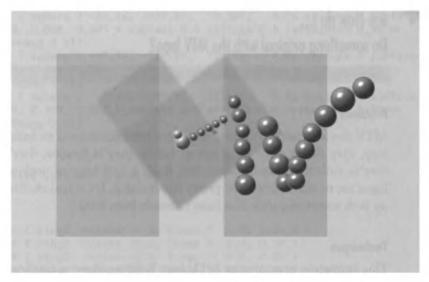


Figure 6-9 The MTV logo: the M is composed of four boxes; the TV was captured interactively with a CAD package

eventually got defined by zooming in, checking alignment, scaling, shifting... no one said this had to be elegant. All it has to do is work.

The following code will draw Figure 6-9 and write a Polyray file containing the MTV logo:

```
DECLARE SUB rotate (x, y, z)
COMMON SHARED rad
DIM red(16), green(16), blue(16), box(2, 4)
SCREEN 12
pi = 3.14159
rad = pi / 180
WINDOW (-160, -80)-(160, 160)
'WINDOW (20, 108)-(52, 140)
'WINDOW (-40, 108)-(0, 140)
'WINDOW (-16, -8)-(16, 16)
' T
DATA -45.17,
              98.32, 0.00,
                              1.58
DATA -43.56,
              96.21, 0.00,
                             3.04
DATA -39.33,
             98.12, 0.00,
                             1.37
DATA -36.91,
             100.03, 0.00,
                              1.26
DATA -34.50,
             101.74, 0.00,
                             1.25
DATA -32.00, 103.51, 0.00,
                             1.22
DATA -29.35, 105.10, 0.00,
                             1.58
DATA -26.05, 106.96, 0.00,
                             1.97
```

```
DATA -21.53,
              108.94, 0.00, 2.21
DATA -17.44,
                             2.68
              111.19, 0.00,
DATA -11.71,
              113.71, 0.00,
                             3.28
DATA -5.80,
              115.95,
                      0.00,
                              3.94
DATA -23.36,
              111.09, 0.00,
                              3.37
DATA -21.25,
              104.37, 0.00,
                              3.71
DATA -20.19,
               97.43,
                              3.33
                       0.00,
DATA -20.26,
               89.66, 0.00,
                              4.31
DATA -19.51,
               80.37, 0.00,
                              4.95
DATA -19.28,
               70.57,
                      0.00,
                              5.64
DATA -8.50,
               99.92, 0.00,
                              5.16
     -7.66,
               90.79, 0.00,
                              4.49
DATA
               81.97, 0.00,
     -4.19,
                              4.72
DATA
DATA
     -2.76,
               73.74, 0.00,
                              4.00
DATA
      1.55,
               74.05,
                      0.00,
                              4.00
               80.15,
DATA
       2.01,
                      0.00,
                              4.00
DATA
       2.86,
               86.94, 0.00,
                              4.48
               95.39,
                              4.80
DATA
      6.90,
                      0.00,
                              5.31
DATA
     12.41,
              104.00,
                      0.00,
DATA
     19.59,
              112.15, 0.00,
                              5.55
              118.72, 0.00,
DATA
     26.75,
                             5.12
s = 2
xt = 95
yt = -140
OPEN "TV.PI" FOR OUTPUT AS #1
FOR p = 1 TO 29
       READ x, y, z, r
       x = x * s
       y = y * s
       r = r * s
       CIRCLE (x + xt, y + yt), r, 3
       PRINT #1, USING "object { sphere <###.###,###.###,###.###>, ##.## ←
_shiny__coral }"; x + xt, y + yt, 0, r
NEXT p
CLOSE #1
DATA -100.00, 130.00,
                       0.00
DATA -100.00 ,-40.00,
                       0.00
DATA -10.00, -40.00,
                       0.00
     -10.00, 30.00,
                       0.00
DATA
DATA
       15.00,
              5.00,
                       0.00
DATA
       40.00, 30.00,
                      0.00
       40.00, -40.00,
DATA
                       0.00
DATA
      130.00, -40.00,
                       0.00
      130.00, 130.00,
                       0.00
DATA
DATA
       40.00, 130.00,
                       0.00
DATA
       15.00, 110.00, 0.00
     -10.00, 130.00, 0.00
DATA
```

continued on next page

```
continued from previous page
DATA -100.00, 130.00, 0.00
s2 = .95
FOR n = 1 TO 13
       READ x, y, z
       x = x * s2
       y = y * s2
       z = z * s2
       IF n = 1 THEN
               PSET(x, y)
       ELSE
               LINE -(x, y)
       END IF
       'LOCATE 1, 1: PRINT USING "###.## "; x, y
       'DO WHILE INKEY$ = "": LOOP
NEXT n
DATA -100,-40,-10,130
DATA 40,-40,130,130
FOR x = 1 TO 2
 READ a, b, c, d
  a = a * s2
  b = b * s2
  c = c * s2
  d = d * s2
  LINE (a, b)-(c, d), B
NEXT x
DATA -52.3,-35,26.4,65
DATA -35,-52.3,2,65
FOR x = 1 TO 2
READ a, b, tx, ty
e = -a * s2
f = -b * s2
c = e * s2
d = b * s2
g = a * s2
h = f * s2
CALL rotate(a, b, 0)
CALL rotate(c, d, 0)
CALL rotate(e, f, 0)
CALL rotate(g, h, 0)
LINE (a + tx, b + ty)-(c + tx, d + ty), x
LINE (c + tx, d + ty)-(e + tx, f + ty), x
LINE (e + tx, f + ty)-(g + tx, h + ty), x
LINE (g + tx, h + ty)-(a + tx, b + ty), x
NEXT x
SUB rotate (x, y, z)
```

```
xrotate = 0
yrotate = 0
zrotate = 45
x0 = x
y0 = y
z0 = z
x1 = x0
y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)
x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
y2 = y1
z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)
x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)
y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)
z3 = z2
x = x3
y = y3
z = z3
```

END SUB

We now have the logo. The next step is to create a spiraling mass of spheres orbiting it. The following program displays spirals, allowing us to vary the thread spacing and sphere motion until we achieve what we want. The part of the code dealing with drawing the cube and camera in Figure 6-10 has been omitted, although the full listing is on the disk. It carries on for quite a while, but it's the spheres we're really interested in here.

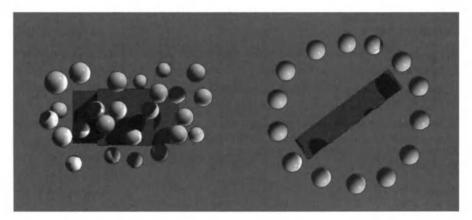


Figure 6-10 The spiraling balls surrounding the MTV logo (greatly simplified)

```
DIM red(16), green(16), blue(16), balls(48, 3), ball2(48, 3), filename$(40)
SCREEN 12
WINDOW (-150, -150)-(490, 250)
FOR Y = 1 TO 4
  FOR x = 1 TO 4
     colornum = x + ((Y - 1) * 4) - 1
     READ red(colornum), green(colornum), blue(colornum)
     KOLOR = 65536 * blue(colornum) + 256 * green(colornum) + red(colornum)
     PALETTE colornum, KOLOR
     COLOR colornum
   NEXT x
NEXT Y
'rainbow palette
DATA 0, 0, 0
DATA 32, 0, 0
DATA 42, 0, 0
DATA 58, 16, 0
DATA 63, 32, 0
DATA 58, 56, 0
DATA 16, 42, 0
DATA 0, 30, 36
DATA 0, 20, 40
DATA 0, 10, 48
DATA 0, 0, 63
DATA 20, 0, 53
DATA 23, 0, 29
DATA 19, 7, 17
DATA 50, 40, 45
DATA 63, 63, 63
pi = 3.1415926536#
radians = 180 / pi
xtran2 = 300
ytran2 = 0
ztran2 = 50
'animation controls
steps = 40 ' number of frames
segments = 12.5 ' spheres per rotation
marque = 2 ' spheres to traverse
thread = 60 'y-distance per rotation
xshow = 1
yshow = 2
zshow = 3
```

```
FOR i = 1 \text{ TO } 36
    balls(i, 1) = 100 * SIN(2 * pi * (posit + (i - 2)) / segments)
    balls(i, 2) = 5 * (i - 4)
    balls(i, 3) = 100 * COS(2 * pi * (posit + (i - 2)) / segments)
NEXT i
fin = 0
FOR posit = 0 TO marque STEP (marque / steps)
   fin = fin + 1
   FOR i = 1 TO 36 'the collection
' undraw 'em
   CIRCLE (balls(i, xshow), balls(i, yshow)), 15, 0
   CIRCLE (ball2(i, xshow), ball2(i, zshow)), 15, 0
   balls(i, 1) = 100 * SIN(2 * pi * (posit + i - 2) / segments)
   balls(i, 3) = 100 * COS(2 * pi * (posit + i - 2) / segments)
   balls(i, 2) = thread / segments * (posit + i - 4)
   ball2(i, 1) = balls(i, 1) + xtran2
   ball2(i, 2) = balls(i, 2) + ytran2
   ball2(i, 3) = balls(i, 3) + ztran2
   col = INT(((balls(i, zshow) / 16) + 9) / 2)
' draw 'em in their new position
   CIRCLE (balls(i, xshow), balls(i, yshow)), 15, col
   CIRCLE (ball2(i, xshow), ball2(i, zshow)), 15, col
   NEXT i
NEXT posit
```

How It Works

Based on this visualization, we've decided to go with a spiral that has a non-integer number of balls (*segments* = 12.5). It prevents the successive loops from lining up. We'll also make the spheres throb so that they occasionally give an unobstructed view of the logo by periodically changing their diameters. We used a mirror texture on these balls. John Hammerton recommended using glass, which made a very nice image but took a long time to render. This spiral is generated using the following code:

```
OPEN "spr" FOR OUTPUT AS #1
PRINT #1, "// now, a spiral"
PRINT #1, "define ring1 200 + 50 * sin(2 * pi * index)"
PRINT #1, "define ring2 200 - 50 * sin(2 * pi * index)"
```

continued on next page

CHAPTER SIX

```
continued from previous page
PRINT #1,
PRINT #1, "define segments 12.5 // spheres per rotation"
PRINT #1, "define marque 2
                                       // spheres to traverse"
PRINT #1, "define thread 60
                                       // y-distance per rotation"
PRINT #1, "define posit marque*index"
PRINT #1,
FOR x = 1 TO 46
   n = RIGHT$("00" + LTRIM$(STR$(x)), 2)
   bx$ = "bx" + n$
   by$ = "by" + n$
   bz$ = "bz" + n$
IF x MOD 2 = 0 THEN
   PRINT #1, "define i "; n$
   PRINT #1, USING "define \ \ ring2 * sin(2 * pi * (posit + (i - 2)) / \Leftarrow
segments)"; bx$
   PRINT #1, USING "define \ \ thread /segments * (posit + i - 4)"; by$
   PRINT #1, USING "define \ \ ring2 * cos(2 * pi * (posit + (i - 2)) / \Leftarrow
segments)"; bz$
ELSE
   PRINT #1, "define i "; n$
   PRINT #1, USING "define \ \ ring1 * \sin(2 * pi * (posit + (i - 2)) / \leftarrow
segments)"; bx$
   PRINT #1, USING "define \ \ thread /segments * (posit + i - 4)"; by$
   PRINT #1, USING "define \ \ ring1 * cos(2 * pi * (posit + (i - 2)) / \Leftarrow
segments)"; bz$
END IF
PRINT #1,
NEXT x
PRINT #1,
PRINT #1, "define sph_size1 15+10*sin(2*pi*index)"
PRINT #1, "define sph_size2 15-10*sin(2*pi*index)"
PRINT #1,
PRINT #1, "define coil"
PRINT #1, "object {"
FOR x = 1 TO 46
  n = RIGHT$("00" + LTRIM$(STR$(x)), 2)
  bx$ = "bx" + n$
   by$ = "by" + n$
   bz$ = "bz" + n$
IF x = 1 THEN
   PRINT #1, USING " object { sphere <\ \,\ \,\ \+75>, sph_size1 }"; ←
bx$, by$, bz$
ELSE
   IF x MOD 2 = 0 THEN
      PRINT #1, USING " + object { sphere <\ \,\ \,\ \+75>, sph_size2 }"; ←
bx$, by$, bz$
```

The spiral creation code uses the data collected during the QuickBasic simulations to calculate the positions of the orbiting spiral loop. This code also contains some of the modifications we'll discuss following the Polyray program listing, illustrating how simple global modifications can be if done within programs.

The rest of the MTV file is assembled and shown in MTV.PI. The logo rides above a rippling yellow sea.

```
// MTV.PI - The Infamous Music Television Logo
start_frame O
end_frame 29
total frames 30
define index frame/total_frames
outfile MTV
include "PLY\COLORS.INC"
viewpoint {
   from <200,180,-350>
   at <0,40,0>
   up <0,1,0>
   angle 40
   resolution 320,200
   aspect 1.433
background SkyBlue
light 0.6 * white, <-50,300,-500>
light 0.6 * white, < 50,300,-500>
define M
object {
   object { box <-100,-40,0>*0.95,<-10,130,150>*0.95 }
 + object { box <40,-40,0>*0.95,<130,130,150>*0.95 }
 + object {
      box <-52.3,-35,0>*0.95,<52.3,35,150>*0.95
      rotate <0,0,45>
      translate <26.4,65,0>
```

continued on next page

```
continued from previous page
 + object {
      box <-35,-52.3,0>*0.95,<35,52.3,150>*0.95
      rotate <0,0,45>
      translate <2,65,0>
   }
   shiny_blue
}
define T
object {
   blob 1.0:
      cylinder <4.6600,56.6400,0.0000>,<7.8800, 52.4200, 0.0000>, 3.16, 10,
      cylinder < 7.8800, 52.4200, 0.0000>,< 16.3400, 56.2400, 0.0000>, 6.08, 10,
      cylinder < 16.3400, 56.2400, 0.0000>,< 21.1800, 60.0600, 0.0000>, 2.74, 10,
      cylinder < 21.1800, 60.0600, 0.0000>,< 26.0000, 63.4800, 0.0000>, 2.52, 10,
      cylinder < 26.0000, 63.4800, 0.0000>,< 31.0000, 67.0200, 0.0000>, 2.50, 10,
      cylinder < 31.0000, 67.0200, 0.0000>,< 36.3000, 70.2000, 0.0000>, 2.44, 10,
      cylinder < 36.3000, 70.2000, 0.0000>,< 42.9000, 73.9200, 0.0000>, 3.16, 10,
      cylinder < 42.9000, 73.9200, 0.0000>,< 51.9400, 77.8800, 0.0000>, 3.94, 10,
      cylinder < 51.9400, 77.8800, 0.0000>,< 60.1200, 82.3800, 0.0000>, 4.42, 10,
      cylinder < 60.1200, 82.3800, 0.0000>,< 71.5800, 87.4200, 0.0000>, 5.36, 10,
      cylinder < 71.5800, 87.4200, 0.0000>,< 83.4000, 91.9000, 0.0000>, 6.56, 10,
      sphere < 83.4000, 91.9000, 0.0000>, 7.88, 10,
      sphere < 48.2800, 82.1800, 0.0000>, 6.74, 10,
      cylinder < 48.2800, 82.1800, 0.0000>,< 52.5000, 68.7400,
                                                                 0.0000 > 7.42, 8,
      cylinder < 52.5000, 68.7400, 0.0000>,< 54.6150, 54.8600, 0.0000>,6.66, 8,
      cylinder < 54.6150, 54.8600, 0.0000>,< 54.4800, 39.3150,
                                                                 0.0000 > 8.62, 8,
      cylinder < 54.4800, 39.3150, 0.0000>, < 55.9800, 15.7400, 0.0000>,9.90, 8,
      cylinder < 55.9800, 15.7400, 0.0000>,< 56.4400, 1.1400, 0.0000>,11.28,8,
      sphere < 55.9800, 15.7400, 0.0000>, 9.9,10,
      sphere < 56.4400, 1.1400, 0.0000>, 11.28, 10
   reflective_cyan
}
define V
object {
   blob 1.0:
      sphere < 78.0000, 59.8400, 0.0000>, 10.32, 10,
      cylinder < 78.0000, 59.8400, 0.0000>, < 79.6800, 41.5800, ←
0.0000 > 10.32, 10,
      cylinder < 79.6800, 41.5800, 0.0000>, < 86.6200, 23.9400, 0.0000>, \( \)
8.98, 10,
      cylinder < 86.6200, 23.9400, 0.0000>, < 89.4800, 7.4800, 0.0000>, \equiv \text{}
9.44, 10,
      cylinder < 89.4800, 7.4800, 0.0000>, < 98.1000, 8.1000, 0.0000>, \equiv \text{}
8.00, 10,
      cylinder < 98.1000, 8.1000, 0.0000>, < 99.0200, 20.3000, 0.0000>, \( \)
8.00, 10,
      cylinder < 99.0200, 20.3000, 0.0000>, <100.7200, 33.8800, 0.0000>, \Leftarrow
      cylinder <100.7200, 33.8800, 0.0000>, <108.8000, 50.7800, 0.0000>, \Leftarrow
8.96, 10,
```

```
cylinder <108.8000, 50.7800, 0.0000>, <119.8200, 68.0000, 0.0000>, \Leftarrow
9.60, 10,
      cylinder <119.8200, 68.0000, 0.0000>, <134.1800, 84.3000, \Leftarrow
0.0000>,10.62, 10,
      cylinder <134.1800, 84.3000, 0.0000>, <148.5000, 97.4400, 0.0000>, ←
10.24, 10,
      sphere <148.5000, 97.4400, 0.0000>, 10.24, 10
   reflective_cyan
}
define pi 3.14159
define rad pi/180
define yellow_ripple
texture {
   noise surface {
      color <1.5,0.75,0.3>
      normal 2
      frequency 1
      phase -index*2*pi
      bump_scale 2
      ambient 0.3
      diffuse 0.4
      specular darkslateblue, 0.7
      reflection 0.5
      microfacet Reitz 10
      }
   scale <20,20,20>
object { disc <0,-40,0>,<0,1,0>,5000 yellow_ripple }
object { cylinder <0,-50,0>,<0,500,0>,5000 matte_blue }
MTV
// now, a spiral
define segments 12.5 // spheres per rotation
define marque 2
                            // spheres to traverse
define thread 60
                            // y-distance per rotation
define posit marque*index
define i 01
define bx01 150 * sin(2 * pi * (posit + (i - 2)) / segments)
define by01 thread /segments * (posit + i - 4)
define bz01 150 * cos(2 * pi * (posit + (i - 2)) / segments)
define i 02
define bx02 150 * sin(2 * pi * (posit + (i - 2)) / segments)
define by 02 thread /segments * (posit + i - 4)
define bz02 150 * cos(2 * pi * (posit + (i - 2)) / segments)
```

continued on next page

```
continued from previous page
define i 46
define bx46 150 * sin(2 * pi * (posit + (i - 2)) / segments)
define by46 thread /segments * (posit + i - 4)
define bz46 150 * cos(2 * pi * (posit + (i - 2)) / segments)
// spiraling spheres
define sph_size 15 + 5*sin(2*pi*index)
object { sphere <bx01,by01,bz01+75>, sph_size matte_white }
object { sphere <bx02,by02,bz02+75>, sph_size matte_white }
...
object { sphere <bx46,by46,bz46+75>, sph_size matte_white }
```

The M is the easiest object here to generate. We used the coordinates and rotational angles from the QuickBasic display code to give us exactly what we needed here. The parts for TV were initially defined as spheres, for use with Truman Brown's Connect the Dots (CTDS) utility. His program is great for piping together solid letters when all you have are control points. We've decided to use blobs here, controlled by a combination of cylindrical and spherical control elements to generate these amorphous letters.

We create a rippling yellow sea texture, place it on a disc, and fence it in with a large blue cylinder that hides a bright blue sky at the horizon. This gives a darker feel to the distance, while allowing other elements in the scene to reflect a bright sky, now hidden from view.

The spiraling sphere elements (a total of 46) are created using data generated using the spiral simulation code, and the sizes of the spheres are made to vary as the spiral rotates around the logo.

Variations

This animation seems to invite modifications, perhaps even attract them. The disk contains seven separate versions of this basic animation, each with something different. The original spiral has a radius of 150 units:

```
define i 01 define bx01 150 * \sin(2 * pi * (posit + (i - 2)) / segments) define by01 thread /segments * (posit + i - 4) define bz01 150 * \cos(2 * pi * (posit + (i - 2)) / segments)
```

We can make the radius of every other sphere breathe in and out by defining *ring1* and *ring2* to vary during the course of the animation:

```
// now, a spiral
define ring1 200 + 50 * sin(2 * pi * index)
define ring2 200 - 50 * sin(2 * pi * index)
```

```
define i 01 define bx01 ring1 * \sin(2 * pi * (posit + (i - 2)) / segments) define by01 thread /segments * (posit + i - 4) define bz01 ring1 * \cos(2 * pi * (posit + (i - 2)) / segments) define i 02 define bx02 ring2 * \sin(2 * pi * (posit + (i - 2)) / segments) define by02 thread /segments * (posit + i - 4) define bz02 ring2 * \cos(2 * pi * (posit + (i - 2)) / segments)
```

The spheres change size as they rotate. We can vary the minimum and maximum size of these spheres so that they obscure varying amounts of the logo.

```
define sph_size 15+15*sin(2*pi*index)
define sph_size 25+10*sin(2*pi*index)
```

They also don't have to change size at the same time. We can define two sphere sizes, sph_size1 and sph_size2 , and oscillate them opposite to one another. We then use the different sphere sizes on every other sphere:

```
define sph_size1 15+15*sin(2*pi*index)
define sph_size2 15-15*sin(2*pi*index)

object { sphere <bx01,by01,bz01+75>, sph_size1 }
object { sphere <bx02,by02,bz02+75>, sph_size2 }
object { sphere <bx03,by03,bz03+75>, sph_size1 }
```

We can render the spheres as mirrors or glass. We can even flip between these two every other frame by first defining the 46 spheres in our spiral as *coil*

```
object coil {
  object { sphere <bx01,by01,bz01+75>, sph_size1 }
+ object { sphere <bx02,by02,bz02+75>, sph_size2 }
+ object { sphere <bx03,by03,bz03+75>, sph_size1 }
...
```

and then calling *coil* and assigning the texture depending on the frame count. Here, *fmod(frame,2)* will be equal to zero only on even frames, so our spheres will be glass on even frames and mirrors on odd ones:

```
if (fmod(frame,2)==0)
  coil {glass }
else
  coil {mirror}
```

We can make the MTV logo itself rock back and forth and bob up and down, by defining precessional angles and displacements. The *bob* variable here is

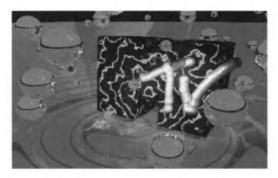


Figure 6-11 MTV logo rocks

```
define precessx 5*sin(4*pi*index)
define precessz 5*cos(4*pi*index)
define bob 10 * (sin(2*pi*index)-cos(4*pi*index))

define logo
object {
    M + T + V
    rotate precessx,0,precessz>
    translate<0,bob,0>
}
```

Figure 6-11 shows the logo bobbing on our yellow sea.



6.5 How do I...

Cover a sphere evenly with other spheres and make it twinkle?

You'll find the code for this in: PLY\CHAPTER6\TWINKLE

Problem

You'd be surprised how often the need arises for a sphere that's been evenly covered by points. There's surface retriangulation, 3-D morph reference points, and bug eyes, just to mention a few instances. When you need an evenly covered sphere, you usually need it bad.

Technique

Let's go over a few techniques to consider our options. The first method is blind guessing. With RANDOME.BAS, we generate lots of random numbers,

convert these to polar coordinates that map onto a sphere, then test each new point to see if it's too close to one that's already there. If it is, we chuck it. If it's not, we keep it and move on. It works, but it's slow and the surface coverage is not very even:

```
'RANDOME.BAS
DECLARE SUB rotate (d, e, f)
TYPE vector
   x AS SINGLE
   y AS SINGLE
   z AS SINGLE
END TYPE
DIM red(16), green(16), blue(16), v(5000) AS vector
COMMON SHARED rad, xrotate, yrotate
SCREEN 12
FOR y = 1 TO 4
   FOR x = 1 TO 4
      colornum = x + ((y - 1) * 4) - 1
      READ red(colornum), green(colornum), blue(colornum)
      col = 65536 * blue(colornum) + 256 * green(colornum) + red(colornum)
      PALETTE colornum, col
      COLOR colornum
   NEXT x
NEXT y
'rainbow palette
DATA 0, 0, 0
DATA 32, 0, 0
DATA 42, 0, 0
DATA 58, 16, 0
DATA 63, 32, 0
DATA 58, 56, 0
DATA 16, 42, 0
DATA 0, 30, 36
DATA 0, 20, 40
DATA 0, 10, 48
DATA 0, 0, 63
DATA 20, 0, 53
DATA 23, 0, 29
DATA 19, 7, 17
DATA 50, 40, 45
DATA 63, 63, 63
pi = 3.13159
rad = pi / 180
min = .05
min2 = min * min
diam = min / 2
```

CHAPTER SIX

```
continued from previous page
LOCATE 2, 1: PRINT min2
DIM kolor(18, 18)
WINDOW (-1.6, -1.2)-(1.6, 1.2)
LINE (-1, -1)-(1, 1), B
LINE (0, 0)-(\min, \min), B
FOR x = 1 TO 18
  FOR y = 1 TO 18
      kolor(x, y) = INT(16 * RND(3))
   NEXT y
NEXT x
OPEN "angles" FOR OUTPUT AS #2
v(1).x = 1
v(1).y = 0
v(1).z = 0
n = 2
DO WHILE n < 5000
 xrotate = 360 * RND(1)
 yrotate = 360 * RND(2)
 zrotate = 360 * RND(3)
 d = 0: e = 0: f = 1
  CALL rotate(d, e, f)
    v(n).x = d
    v(n).y = e
    v(n).z = f
    FOR m = 1 TO n - 1
       d2 = (v(m).x - v(n).x) ^2 + (v(m).y - v(n).y) ^2 + (v(m).z - v(n).z) ^2
        IF d2 < min2 THEN GOTO runaway
    NEXT m
    IF f > 0 THEN
    CIRCLE (d, e), diam, (2 + (f + 1) * 12) MOD 16
    PRINT #2, USING "###.#### ###.#### ###.###"; d, e, f
  n = n + 1
  LOCATE 1, 1: PRINT n
  LINE (-1, 1.1)-(2, 1.1), 0
  t = 0
runaway:
  t = t + 1
  LOCATE 1, 1: PRINT n
  LINE (-1, 1.1)-(-1 + LOG(t / 200), 1.1)
  L00P
  CLOSE #2
```

```
SUB rotate (d, e, f)
   x0 = d
   y0 = e
   z0 = f
   x1 = x0
   y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
   z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)
   x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
   y2 = y1
   z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)
   x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)
   y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)
   z3 = z2
   d = x3
    e = y3
    f = z3
END SUB
```

Figure 6-12 shows the type of coverage we get with this approach.

A better approach, but even slower than the first, uses a tiny step size and creeps over the surface of a sphere, cautiously putting a new sphere in the first place it finds that will accommodate it. This is shown in the following listing (SPH2.BAS); even on a fast machine, though, it runs overnight.

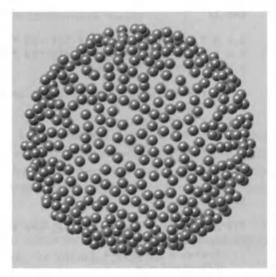


Figure 6-12 Random quessing sphere coverage

```
' SPH2.BAS
DECLARE SUB rotate (x, y, z)
COMMON SHARED rad
n = 2000
DIM x(n), y(n), z(n)
OPEN "dome" FOR OUTPUT AS #1
PRINT #1, 1000
SCREEN 12
WINDOW (-1.6, -1.2)-(1.6, 1.2)
pi = 3.14159
rad = pi / 180
R = 1
CIRCLE (0, 0), R
x(1) = 0
y(1) = 0
z(1) = 1
circles = 1
target = .1
rate = 1
CIRCLE (x(circles), y(circles)), target / 2
DO WHILE c > -.995
  ' focus on the rate increase
  phi = phi + 1
   IF phi > 360 THEN
     phi = 0
      theta = theta + rate
   END IF
  a = R * SIN(rad * theta) * COS(rad * phi)
   b = R * SIN(rad * theta) * SIN(rad * phi)
   c = R * COS(rad * theta)
   'find the distance to the closest sphere already drawn
   min = 100
   FOR n = 1 TO circles
     d = ((a - x(n)) ^2 + (b - y(n)) ^2 + (c - z(n)) ^2) ^.5
      IF d < min THEN
         min = d
         p = n
      END IF
   NEXT n
   'if none of 'em are too close, add this one to the bunch
   IF min > target THEN
     circles = circles + 1
     x(circles) = a
```

```
y(circles) = b
      z(circles) = c
      CIRCLE (x(circles), y(circles)), target / 2
     LINE (x(circles), y(circles))-(x(p), y(p)), 12
     PRINT #1, USING "##.##### "; a, b, c
     LOCATE 1, 1: PRINT circles, c
      CIRCLE (0, 0), (a^2 + b^2).
  END IF
L00P
CLOSE #1
SUB rotate (x, y, z)
    xrotate = 90
    yrotate = 0
    zrotate = 0
     x0 = x
     y0 = y
     z0 = z
     x1 = x0
     y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
     z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)
     x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
     y2 = y1
     z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)
     x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)
     y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)
      z3 = z2
     x = x3
      y = y3
      z = z3
```

END SUB

The results, shown in Figure 6-13, look much better than the previous version. This code takes a long time to generate with QuickBasic. A faster version was done in C by Eric Deren, and the executable is included on the disk. There's also a faster subdivision version by Jon Leech which has been included as an executable.

This is a long way to go to get the data for a simple animation, but it's worth it. SPH2 generates a list of sphere locations, which are converted into a Polyray compatible form by using the following program:

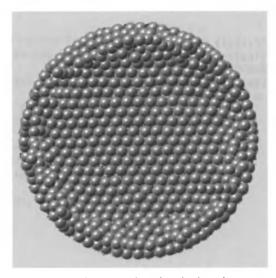


Figure 6-13 Sphere covered evenly with other spheres

```
OPEN "dome" FOR INPUT AS #1
OPEN "d:\ply\dat\hex\dome.pi" FOR OUTPUT AS #2
FOR x = 1 TO 642
    INPUT #1, a, b, c
    PRINT #2, USING "object { sphere <##.######, ##.######, ##.#####>, 0.1 
matte__white }"; a, b, c
NEXT x
CLOSE #1
CLOSE #2
```

This array of spheres is enclosed inside a larger sphere, and spotlights are placed around it in the following listing (DOME.PI). A combination of blue and coral lights above gives it a Maxfield Parrish feel. We rotate the mass and shrink the sizes of each sphere until they only catch the rays occasionally. With the darkness below and lighting from above, the sparkling is striking:

```
// DOME.PI - Semi Sorta Geodesic Dome Type Deal
start_frame 0
end_frame 179
total_frames 180
outfile "dome"
include "\PLY\COLORS.INC"
viewpoint {
   from <-3,0,-3> //-5,0,-3
   at <0,0,3>
```

```
up <0,0,1>
   angle 70
   resolution 480,480 // 160,480
   aspect 1
                    // 0.33
spot_light < 2, 1, 0>,<-5,-5, 5>,<0,0,0>,3,15,30
spot_light < 0.5, 0, 1>,< 5,-5, 5>,<0,0,0>,3,15,30
spot_light < 0, 0, 0.5>,< 5, 5, 5>,<0,0,0>,3,15,30
spot_light < 0.25, 0, 1>,<-5, 5, 5>,<0,0,0>,3,15,30
spot_light < 0.25, 0.5, 1>,<0, 0, -5>,<0,0,0>,3,5,20
define bright texture { matte { color white*2} }
object { sphere <0.0,0.0,0.5>, 9 bright }
define pi 3.14159
define rad pi/180
// shrink and rotate
define radius 0.1*(1.1+cos(rad*frame*2))/2
object {
   object { sphere < 0.101056, 0.000000, 0.994881>, radius matte_white }
 + object { sphere < 0.050528,
                               0.087517,
                                          0.994881>, radius matte_white }
+ object { sphere <-0.050528, 0.087517,
                                          0.994881>, radius matte_white }
 + object { sphere <-0.101056, 0.000000,
                                          0.994881>, radius matte_white }
 + object { sphere <-0.050528, -0.087517,
                                          0.994881>, radius matte white }
 + object { sphere < 0.050528, -0.087517,
                                          0.994881>, radius matte_white }
+ object { sphere < 0.150384, 0.086824,
                                          0.984808>, radius matte_white }
 + object { sphere < 0.000000, 0.173648,
                                          0.984808>, radius matte white }
 + object { sphere <-0.150383, 0.086824,
                                          0.984808>, radius matte_white }
 + object { sphere <-0.150384, -0.086824,
                                          0.984808>, radius matte_white }
+ object { sphere <-0.000001, -0.173648, 0.984808>, radius matte_white }
 + object { sphere < 0.150383, -0.086825, 0.984808>, radius matte_white }
 + object { sphere < 0.201078, 0.000000, 0.979575>, radius matte white }
 + object { sphere < 0.100539, 0.174139, 0.979575>, radius matte_white }
(600 sphere)
 + object { sphere <-0.325566, 0.945518, -0.001730>, radius matte_white }
 + object { sphere <-0.981625, 0.190811, -0.001730>, radius matte_white }
 + object { sphere <-0.656060, -0.754706, -0.001730>, radius matte_white }
+ object { sphere < 0.325564, -0.945518, -0.001730>, radius matte_white }
 + object { sphere < 0.981625, -0.190813, -0.001730>, radius matte_white }
   rotate <0,0,frame*2>
```

How It Works

We're inside a large sphere and below this dome. There are four very brightly colored spotlights shining on our dome, using Maxfield Parrish colors for mood:

```
spot_light < 2, 1, 0>,<-5,-5, 5>,<0,0,0>,3,15,30
spot_light < 0.5, 0, 1>,< 5,-5, 5>,<0,0,0>,3,15,30
spot_light < 0, 0, 0.5>,< 5, 5, 5>,<0,0,0>,3,15,30
spot_light < 0.25, 0, 1>,<-5, 5, 5>,<0,0,0>,3,15,30
```

One spotlight is directly below the sphere dome, shining up:

```
spot_light < 0.25, 0.5, 1>,<0, 0, -5>,<0,0,0>,3,5,20
```

The last line in the sphere dome definition makes the whole thing rotate. The beauty of this animation is that, other than the rotation, it really only has one other moving part. The frame-dependent expression

```
// shrink and rotate
define radius 0.1*(1.1+cos(rad*frame*2))/2
```

sets the radius for all the spheres in the dome to vary between 0.05 and 1.15. Without antialiasing, 0.05 is so small that the ray tracer will only randomly register that they're there, which makes them twinkle (see Figure 6-14).



Figure 6-14 Twinkle twinkle little sphere

CHAPTER

BLOBS

lobs are an affectionate name given to objects that, for lack of a better description look like, well... blobs. They are built out of any number of smaller units called metaballs, each of which is defined by a location, a radius, and a strength. Each metaball generates a field that defines a value for every point in the space surrounding it. The size of this field is set by its radius; the magnitude of this field is set by its strength. When it comes time to render a surface, the field strength contributions for all the metaballs at any particular point in 3D space are added together, and wherever this value equals the threshold value, a surface element appears. Some examples are shown in Figure 7-1.

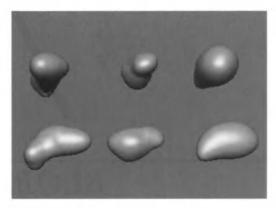


Figure 7-1 Blobs

There are presently two valid methods for specifying blobs in Polyray. The "new" way goes like this:

```
blob threshold:
sphere <x, y, z>, strength, radius,
cylinder <x0, y0, z0>, <x1, y1, z1>, strength, radius,
plane <nx, ny, nz>, d, strength, dist,
...
```

Here the threshold defines where the surface forms, and the strength and radius specify the potential field surrounding each controlling element. The "old" way, back in the days when the only type of blob was spherical, went like this:

```
blob threshold:

strength, radius,<x, y, z>,

strength, radius,<x, y, z>,

strength, radius,<x, y, z>,
```

The parser in Polyray (which translates text files you write into meaningful syntax Polyray uses to create the actual images) can deal with both formats, but if you use the new format, you'll have both planar and cylindrical blobs to play with.

The best way to learn about blobs is to play with them. A most remarkable example of what blobs are capable of is contained in the sample files that come with Polyray called SQUISH (see Figure 7-1). This animation uses 16 metaballs, each moving independent of the others. Some orbit, while some slide back and forth. The squirming animation that develops resembles a clump of water in free fall.

Blobs are capable of generating some remarkably lifelike forms, but they do have limitations. They aren't quite as simple to use as modeling clay. You'll have to select the proper strength, radius, and threshold values or else it will end up looking like you're just sticking a lot of spheres together. Also, the more metaballs your blobs have, the longer the rendering times will be.



7.1 How do I...

Visualize potential fields about objects?

You'll find the code for this in: PLY\CHAPTER7\3LEVELS

Problem

Volume rendering is an exciting visualization field. It allows you to see the internal structure of solid objects, as semi-transparent layers or cut-away views. It can be used to illustrate the internal structures of complex objects such as hips and skulls. People interested in such things are usually either recovering from exposure to sudden changes in their inertial reference frames (Editor's note: we think he means "car crashes") or gainfully employed in assisting them. We won't deal with such lofty topics here, since the data sets are enormous and the equipment used to gather them very expensive. With enough data storage capacity and processing power, we could use the following principles to render 3-D volumes. We'll use blobs of various thresholds to simulate our volume data, and move them around to give a good 3-D feel to them.

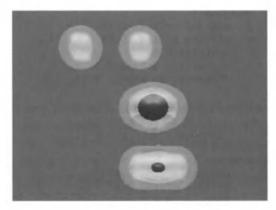


Figure 7-2 Oscillating three-level blob

Technique

Potential fields, like an electric field around an object, can be made visible by rendering surfaces of constant electric potential. Additional 3D information can be imparted by using independent, multilayered, transparent blobs. While metaballs of one blob add together to determine the final shape, separate blobs do not influence each other. The following simple program (3LEVEL1.PI) is a three-layered blob, with one of its metaballs fixed and the other one oscillating back and forth along the *x* axis (Figure 7-2).

```
// 3LEVEL1.PI
start_frame 0
end_frame 30
total_frames 30
outfile 3lev
include "\PLY\COLORS.INC"
viewpoint {
   from <0,0,-8>
   at <0,0,0>
   up <0,1,0>
   angle 30
   resolution 200,100
   aspect 2*1.43/1.6
   }
background SkyBlue
light 0.6 * white, <-15,30,-25>
light 0.6 * white, < 15,30,-25>
define yellow_glass
texture {
   surface {
      ambient yellow, 0.05
      diffuse yellow, 0.05
      specular 0.6
      reflection white, 0.1
      transmission white, 0.95, 1.0
   }
define green_glass
texture {
   surface {
      ambient green, 0.05
      diffuse green, 0.05
      specular 0.2
```

```
reflection white, 0.1
      transmission white, 0.95, 1.0
      }
   }
define blue_glass
texture {
   surface {
      ambient blue, 0.2
      diffuse blue, 0.6
      specular 0.6
      reflection blue, 0.1
   }
define pi 3.14159
define index frame/total_frames
define a 3*sin(2*pi*index)
object {
   blob 0.5:
      1.0, 2.0, <0, 0, 0>,
      1.0, 2.0, <a, 0, 0>
   yellow_glass
}
object {
   blob 0.75:
      0.9, 2.0, <0, 0, 0>,
      0.9, 2.0, < a, 0, 0 >
   green_glass
}
object {
   blob 1.0:
      0.8, 2.0, <0, 0, 0>,
      0.8, 2.0, <a, 0, 0>
   blue_glass
}
```

How It Works

We move one blob element back and forth three units, and give all the blob elements the same radius, but vary the strengths from 1.0 for the outer yellow blob to 0.8 for the inner blue blob. The interior blue glass ball disappears, because it doesn't have enough strength to remain solid once the metaballs defining it move more than about one unit apart. The two outer layers remain visible after they separate because they form at lower thresholds.

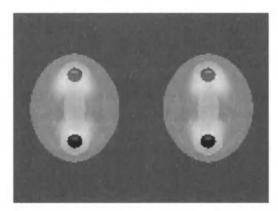


Figure 7-3 Blob cat's cradle

This animation makes harmless screen candy, provided you have low expectations. A more involved layered blob animation in 2D starts with four nested, three-layer blobs. It splits in half left to right, then top to bottom, then rejoins left to right and top to bottom, in a sort of blob cat's cradle (Figure 7-3). 3LEVEL2.PI is the code for this.

```
// 3LEVEL2.PI
start frame 1
end_frame 60
total_frames 60
define index frame/total_frames
outfile 3lev
include "\PLY\COLORS.INC"
viewpoint {
   from rotate(<5,5,-5>,<0,-90*index,0>)
   at <0,0,0>
   up <0,1,0>
   angle 30
   resolution 200,200
   aspect 1*1.433/1.6
background MidnightBlue
light 0.6 * white, <-15,30,-25>
light 0.6 * white, < 15,30,-25>
define yellow_glass
texture {
   surface {
      ambient yellow, 0.05
```

```
diffuse yellow, 0.05
      specular 0.6
      reflection white, 0.1
      transmission white, 0.95, 1.0
      }
   }
define green_glass
texture {
   surface {
      ambient green, 0.05
      diffuse green, 0.05
      specular 0.2
      reflection white, 0.1
      transmission white, 0.95, 1.0
      }
   }
define blue_glass
texture {
   surface {
      ambient blue, 0.2
      diffuse blue, 0.6
      specular 0.6
      reflection blue, 0.1
      }
   }
define pi 3.14159
define index frame/total_frames
if (index >= 0 && index < 0.25) {
   define a (1 - COS(4 * pi * index)) / 2
   define b 0
}
if ((index >= 0.25) && (index < 0.5)) {
   define a 1
   define b (COS(4 * pi * index) + 1) / 2
7
if ((index >= 0.5) && (index < 0.75)) {
   define a (COS(4 * pi * index) + 1) / 2
   define b 1
}
if (index >= 0.75) {
   define a 0
   define b (1 - COS(4 * pi * index)) / 2
define threshold 0.9
define strength 1.0
```

continued on next page

```
continued from previous page
define radius 1.5
object {
   blob threshold:
      sphere <a, b, 0>, strength, radius,
      sphere <a,-b, 0>, strength, radius,
      sphere <-a, b, 0>, strength, radius,
      sphere <-a,-b, 0>, strength, radius,
   yellow_glass
}
define radius 1.0
object {
   blob threshold:
      sphere <a, b, 0>, strength, radius,
      sphere <a,-b, 0>, strength, radius,
      sphere <-a, b, 0>, strength, radius,
      sphere <-a,-b, 0>, strength, radius,
   green_glass
}
define radius 0.5
object {
   blob threshold:
      sphere <a, b, 0>, strength, radius,
      sphere <a,-b, 0>, strength, radius,
      sphere <-a, b, 0>, strength, radius,
      sphere <-a,-b, 0>, strength, radius,
   blue_glass
}
```

3LEVEL2.PI generates this dividing/kneading motion for our four blobs by using four conditional statements that define their motions. The animation is divided into four parts. During the first quarter, we smoothly split the blob

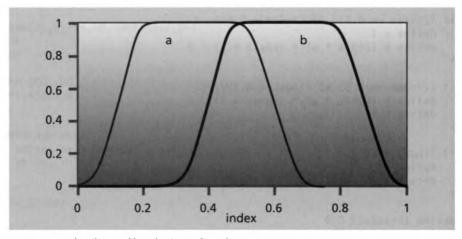


Figure 7-4 Values for a and b set by the conditional statements

into two parts using a cosine spline, by assigning one blob positive values for a, the other one negative values. We split it into four parts during the second quarter, this time using positive and negative values for b. We rejoin the first split during the third quarter, and then collapse the figure back into a single object during the fourth quarter. The values determined by the conditional code are shown in Figure 7-4.

```
if (index >= 0 && index < 0.25) {
   define a (1 - COS(4 * pi * index)) / 2
   define b 0
}
if ((index >= 0.25) && (index < 0.5)) {
   define a 1
   define b (COS(4 * pi * index) + 1) / 2
}
if ((index >= 0.5) && (index < 0.75)) {
   define a (COS(4 * pi * index) + 1) / 2
   define b 1
}
if (index >= 0.75) {
   define a 0
   define b (1 - COS(4 * pi * index)) / 2
}
```

Now we've done a 1-D and a 2-D blob splitting animation. Let's bump it up to 3-D. We'll place it over some textured surface and rotate the viewpoint for some better depth clues. 3LEVEL3.PI is shown here:

```
// 3LEVEL3.PI
start_frame 0
end_frame 179
total_frames 180

define index frame/total_frames
outfile 3lev
include "\PLY\COLORS.INC"

viewpoint {
   from rotate(<5,5,-5>,<0,-360*index,0>)
   at <0,0,0>
   up <0,1,0>
   angle 30
   resolution 200,200
   aspect 1*1.433/1.6
}
```

background MidnightBlue

```
continued from previous page
light 0.6 * white, <-15,30,-25>
light 0.6 * white, < 15,30,-25>
define yellow_glass
texture {
   surface {
      ambient yellow, 0.05
      diffuse yellow, 0.05
      specular 0.6
      reflection white, 0.1
      transmission white, 0.95, 1.0
   }
define green_glass
texture {
   surface {
      ambient green, 0.05
      diffuse green, 0.05
      specular 0.2
      reflection white, 0.1
      transmission white, 0.95, 1.0
      }
   }
define blue_glass
texture {
   surface {
      ambient blue, 0.2
      diffuse blue, 0.6
      specular 0.6
      reflection blue, 0.1
      }
   }
define pi 3.14159
define index frame/total_frames
define a 0.7071+sin(2*pi*index)
define b 0.7071+sin(2*pi*(index+0.33))
define c 0.7071+sin(2*pi*(index+0.66))
define threshold 0.9
define strength 1.0
define radius 1.5
object {
   blob threshold:
      sphere <a, b, c>, strength, radius,
      sphere <a,-b, c>, strength, radius,
      sphere <-a, b, c>, strength, radius,
      sphere <-a,-b, c>, strength, radius,
```

```
sphere <a, b, -c>, strength, radius,
      sphere <a,-b, -c>, strength, radius,
      sphere <-a, b,-c>, strength, radius,
      sphere <-a,-b,-c>, strength, radius
   yellow_glass
}
define radius 1.0
object {
   blob threshold:
      sphere <a, b, c>, strength, radius,
      sphere <a,-b, c>, strength, radius,
      sphere <-a, b, c>, strength, radius,
      sphere <-a,-b, c>, strength, radius,
      sphere <a, b,-c>, strength, radius,
      sphere <a,-b,-c>, strength, radius,
      sphere <-a, b,-c>, strength, radius,
      sphere <-a,-b,-c>, strength, radius
   green_glass
}
define radius 0.5
object {
   blob threshold:
      sphere <a, b, c>, strength, radius,
      sphere <a,-b, c>, strength, radius,
      sphere <-a, b, c>, strength, radius,
      sphere <-a,-b, c>, strength, radius,
      sphere <a, b,-c>, strength, radius,
      sphere <a,-b,-c>, strength, radius,
      sphere <-a, b,-c>, strength, radius,
      sphere <-a,-b,-c>, strength, radius
   blue_glass
}
// yellow dented/wrinkled appearance
define dented_plutonium
texture {
   noise surface {
      color <1.0, 0.5, 0.1>*1.1
      normal 1
      frequency 2
      bump_scale 3
      ambient 0.2
      diffuse 0.5
      specular 0.7
      microfacet Reitz 10
   scale <0.2, 0.2, 0.2>
   }
object {disc <0,-2.5,0>,<0,1,0>,10 dented_plutonium }
```

The output of this animation is shown in Figure 7-5. If you look at Figure 7-4, you see we went to a lot of trouble with conditional statements to make a more or less flattened sine wave. Rather than try conditionals again for three components, we just use three sine waves shifted upward a bit (Figure 7-5).

3LEVEL3.PI rotates the vantage point around our blob as it flies apart then brings it back together. Dented plutonium on a disk provides a suitable backdrop. A scene is shown in Figure 7-6.

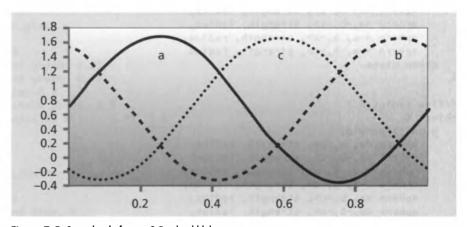


Figure 7-5 Control code for our 3-D orbital blobs

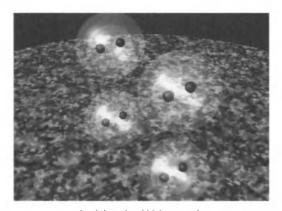


Figure 7-6 Orbital three-level blobs over plutonium



🧩 7.2 How do I...

Create a genetically warped bunch of bananas?

You'll find the code for this in: PLY\CHAPTER7\BANARAMA

Problem

Varying the potential of a blob allows you to billow the shape around the controlling metaballs. Placing a series of metaballs along a spiral path and then varying the potential of the blob makes it resemble one of those long balloons being inflated. Entertainers used to twist them into poodles and horses and bunnies; we're going to make a cubical rotating mass of bananas.

Technique

We start with a cube, rotate a copy of it into an offset position, for a total of 16 vertices. We get 16 spiral blobs, one for each vertex. We tumble both cubes and increase their size as they tumble. Every 10°, we spit out a metaball. This defines a series of metaballs for our bunch (see Figure 7-7). The threshold varies sinusoidally between 0.002 and 0.022 using

define threshold 0.002 + 0.01 * (cos(rad*4*frame)+1)

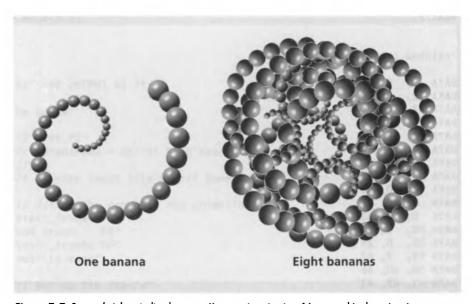


Figure 7-7 One and eight spiraling bananas. Use your imagination: 16 are used in the animation

The following QuickBasic listing creates the data file for us.

```
' Bananarama - QB File Generator
' (c) Jeff Bowermaster, 1992
DECLARE SUB rotate (x, y, z)
COMMON SHARED rad, xrotate, yrotate, zrotate
TYPE Vector
 x AS SINGLE
 y AS SINGLE
 z AS SINGLE
END TYPE
DIM cube(16) AS Vector, last(16) AS Vector
DIM red(16), green(16), blue(16)
' set the screen up with pretty rainbow colors
SCREEN 12
WINDOW (-3.2, -2.4)-(3.2, 2.4)
FOR y = 1 TO 4
   FOR x = 1 TO 4
      colornum = x + ((y - 1) * 4) - 1
      READ red(colornum), green(colornum), blue(colornum)
      KOLOR = 65536 * blue(colornum) + 256 * green(colornum) + red(colornum)
      PALETTE colornum, KOLOR
      COLOR colornum
  NEXT x
NEXT y
'rainbow palette
DATA 0, 0, 0
DATA 32, 0, 0
DATA 42, 0, 0
DATA 58, 16, 0
DATA 63, 32, 0
DATA 58, 56, 0
DATA 16, 42, 0
DATA 0, 30, 36
DATA 0, 20, 40
DATA 0, 10, 48
DATA 0, 0, 63
DATA 20, 0, 53
DATA 23, 0, 29
DATA 19, 7, 17
DATA 50, 40, 45
DATA 63, 63, 63
pi = 3.1415926535#
```

```
rad = pi / 180
' a cube
cube(1).x = 1: cube(1).y = 1: cube(1).z = 1
cube(2).x = 1: cube(2).y = 1: cube(2).z = -1
cube(3).x = 1: cube(3).y = -1: cube(3).z = -1
cube(4).x = 1: cube(4).y = -1: cube(4).z = 1
cube(5).x = -1: cube(5).y = 1: cube(5).z = 1
cube(6).x = -1: cube(6).y = 1: cube(6).z = -1
cube(7).x = -1: cube(7).y = -1: cube(7).z = -1
cube(8).x = -1: cube(8).y = -1: cube(8).z = 1
FOR a = 1 TO 8
   xrotate = 45
   yrotate = 45
   zrotate = 45
'rotate
  x = cube(a).x
   y = cube(a).y
   z = cube(a).z
   CALL rotate(x, y, z)
'give us a second cube at an angle to the first
   cube(a + 8).x = x
   cube(a + 8).y = y
   cube(a + 8).z = z
NEXT a
OPEN "BANA.PI" FOR OUTPUT AS #1
'generate the header
PRINT #1, "// BANA.PI"
PRINT #1, "// Bananarama - Spiral Blob Mass "
PRINT #1, "//"
PRINT #1, "// Polyray input file - Jeff Bowermaster"
PRINT #1,
PRINT #1, "// define the range of the animation"
PRINT #1, "start_frame
PRINT #1, "end_frame
PRINT #1, "total_frames 90"
PRINT #1, "outfile bana"
PRINT #1,
PRINT #1, "// set up the camera"
PRINT #1, "viewpoint {"
```

continued on next page

CHAPTER SEVEN

```
continued from previous page
PRINT #1, " from <3,3,-3>"
PRINT #1, " at <0,0,0>"
PRINT #1, " up <0,1,0>"
PRINT #1, " angle 45"
PRINT #1, " resolution 320,200"
PRINT #1, " aspect 1.43"
PRINT #1, " }"
PRINT #1,
PRINT #1, "// set up background color & lights"
PRINT #1, "background skyblue"
PRINT #1, "light <10,0,-10>"
PRINT #1, "light <-10,0,-10>"
PRINT #1,
PRINT #1, "include "; CHR$(34); "\PLY\COLORS.INC"; CHR$(34)
PRINT #1,
PRINT #1, "define pi 3.14159"
PRINT #1, "define rad pi/180"
PRINT #1,
PRINT #1, "define threshold 0.002 + 0.01 * (cos(rad*4*frame)+1)"
PRINT #1, "define strength 0.01"
PRINT #1, "define range 0.4"
PRINT #1,
PRINT #1, "// fire 1"
PRINT #1,
FOR b = 1 TO 16
   PRINT #1, "object {"
   PRINT #1, " blob threshold:"
stp = 1
maxa = 360
FOR angle = 1 TO maxa STEP stp
   FOR a = 1 TO 16
      xrotate = COS(rad * angle)
      yrotate = SIN(rad * angle)
      zrotate = 1
'rotate
      CALL rotate(cube(a).x, cube(a).y, cube(a).z)
    NEXT a
'generate the metaballs
   IF angle MOD 10 = 0 THEN
      IF angle < maxa THEN
         PRINT #1, USING "strength, range, < ##.####, ##.####, ##.#### >,"; ←
cube(b).x * angle / 360, cube(b).y * angle / 360, cube(b).z * angle / 360
      ELSE
```

```
PRINT #1, USING "strength, range, < ##.####, ##.####, ##.#### >"; ←
cube(b).x * angle / 360, cube(b).y * angle / 360, cube(b).z * angle / 360
      END IF
      CIRCLE (cube(b).x * angle / 360, cube(b).y * angle / 360), angle / \Leftarrow
1800, INT(2 * (cube(b).z + 3))
      last(b) = cube(b)
   END IF
NEXT angle
   PRINT #1, "
                root_solver Ferrari"
   PRINT #1, "
                 u steps 20"
   PRINT #1, "
                 v_steps 20"
   READ texture$
                "; texture$
   PRINT #1, "
   PRINT #1, "
                rotate <0, 4*frame, 0>"
   PRINT #1, " }"
PRINT #1,
NEXT b
CLOSE #1
DATA reflective_grey
DATA reflective_blue
DATA reflective_red
DATA reflective_green
DATA reflective_orange
DATA reflective_yellow
DATA reflective cyan
DATA reflective_brown
DATA reflective_tan
DATA reflective_coral
DATA reflective_gold
DATA shiny_red
DATA reflective_blue
DATA shiny_blue
DATA shiny orange
DATA shiny_yellow
SUB rotate (x, y, z)
    x0 = x
    y0 = y
    z0 = z
    y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
    z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)
    x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
    z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)
                                                                  continued on next page
```

```
continued from previous page
    x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)
    y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)
    z3 = z2
    x = x3
    y = y3
    z = z3
END SUB
           This generates the following Polyray data file:
// BANA.PI
// Bananarama - Spiral Blob Mass
// Polyray input file - Jeff Bowermaster
// define the range of the animation
start_frame 0
end_frame
total_frames 90
outfile bana
// set up the camera
viewpoint {
   from <3,3,-3>
   at <0,0,0>
   up <0,1,0>
   angle 45
   resolution 320,200
   aspect 1.43
// set up background color & lights
background skyblue
light <10,0,-10>
light <-10,0,-10>
include "\ply\colors.inc"
define pi 3.14159
define rad pi/180
define threshold 0.002 + 0.01 * (cos(rad*4*frame)+1)
define strength 0.01
define range 0.4
// fire 1
object {
   blob threshold:
strength, range, < 0.02354, 0.02703, 0.03209 >,
strength, range, < 0.04122, 0.05037, 0.07087 >,
```

```
strength, range, < 0.05771,
                              0.06867,
                                       0.11308 >
strength, range, < 0.07750,
                              0.08327,
                                       0.15523 >
                              0.09792,
strength, range, < 0.10378,
                                       0.19368 > 7
strength, range, < 0.13732,
                              0.11786,
                                       0.22491 >
                             0.14867,
strength, range, < 0.17601,
                                       0.24564 > 
                              0.19489,
strength, range, < 0.21480,
                                        0.25304 >
strength, range, < 0.24625,
                              0.25866,
                                        0.24485 >
strength, range, < 0.26172,
                             0.33879,
                                        0.21955 >
                              0.43020,
strength, range, < 0.25278,
                                        0.17643 > 2
strength, range, < 0.21288,
                             0.52406,
                                        0.11565 >
strength, range, < 0.13890,
                             0.60864,
                                       0.03831 >
strength, range, < 0.03230, 0.67066, -0.05362 >,
strength, range, < -0.10031, 0.69716, -0.15730 >,
strength, range, < -0.24734, 0.67748, -0.26915 >,
strength, range, < -0.39322, 0.60513, -0.38494 >,
strength, range, < -0.51997, 0.47924, -0.49996 >,
strength, range, < -0.60933, 0.30532, -0.60922 >,
strength, range, < -0.64512, 0.09522, -0.70759 >,
strength, range, < -0.61541, -0.13375, -0.79007 >,
strength, range, < -0.51440, -0.36045, -0.85196 >,
strength, range, < -0.34356, -0.56217, -0.88908 >,
strength, range, < -0.11197, -0.71726, -0.89796 >,
strength, range, < 0.16427, -0.80769, -0.87602 >,
strength, range, < 0.46355, -0.82143, -0.82171 >,
strength, range, < 0.76100, -0.75413, -0.73463 >,
strength, range, < 1.03139, -0.61000, -0.61560 >,
strength, range, < 1.25206, -0.40159, -0.46672 >,
strength, range, < 1.40585, -0.14852, -0.29132 >,
strength, range, < 1.48329, 0.12485, -0.09390 >,
strength, range, < 1.48393,
                             0.39236, 0.11995 >,
strength, range, < 1.41651,
                             0.62945,
                                        0.34371 > 
strength, range, < 1.29783,
                             0.81640,
                                        0.57015 >
strength, range, < 1.15053,
                             0.94095,
                                        0.79155 >
strength, range, < 1.00000,
                             1.00005,
                                       0.99995 >
   root_solver Ferrari
   u_steps 20
   v_steps 20
   reflective_grey
   rotate <0, 4*frame, 0>
(repeats for a total of 16 bananas)
```

The resulting image has some really odd motion, and is shown in Figure 7-8.

How It Works

We get QuickBasic to manufacture some oddly shaped intertwining spirals, assign some colorful textures to them, and vary the threshold where the surfaces form to make them inflate and deflate as they rotate.

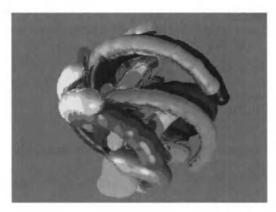


Figure 7-8 Banarama in motion

Comments

A wiggly, particle systems, blob-like octopus animation based on this figure would use about 50% of this book's information in a single animation. Seriously, we should rewrite the code to generate the appendages inside Polyray, and use random motions to make the bunches sway to and fro.



7.3 How do I...

Tumble blob crosses and see how they form?

You'll find the code for this in: PLY\CHAPTER7\STARS

Problem

One of the chief dangers in relying on a bug in ray tracing code to produce some interesting effect is that once it gets repaired, you're out of luck if you try to recreate it. Polyray used to have a problem with blobs, where shifting the value for the strength over a very narrow range (0.00096 ± 0.00005), would cause the blob to shrink down to nothing and vanish in a really interesting fashion. But Alexander fixed that one, so now it takes REALLY extreme values (on the order of say 0.00000001) to make Polyray's solver code upset, and even then it's mostly just roundoff noise. This is where you wander off the edge of numerical accuracy into vague, random regions.

Technique

If we were using the previous version of Polyray, the following animation (STARS.PI) would show tumbling crosses that would shift between being

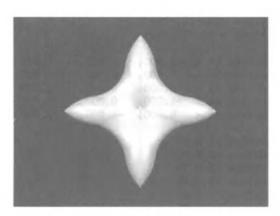


Figure 7-9 A single blob cross

visible and invisible. With the current version, they stay visible all the time. Nevertheless, the following animation illustrates the kinds of shapes blobs form over the range where they differ the most from spheres. It's composed of nine blob crosses, and the values that define them slide back and forth as they tumble.

A single blob cross is shown in Figure 7-9. Blob crosses have interesting dimples in them near the centers.

```
//
// Stars - Blobs with Holes That Rotate
// Polyray input file
// define the range of the animation
start_frame
               0
end_frame
              89
total_frames
              90
outfile tra
// set up the camera
viewpoint {
   from <0,-8.0,-5.0>
   at <0,1,0>
   up <0,1,0>
   angle 45
   resolution 320,200
   aspect 1.433
   }
include "\PLY\COLORS.INC"
// Set up background color & lights
background skyblue
```

CHAPTER SEVEN

```
continued from previous page
light <10,-10,-10>
light <-10,-10,-10>
define b1 <
              0.000,
                        1.000,
                                  0.000 >
                        0.000,
define b2 <
              1.000,
                                  0.000 >
define b3 <
              0.000,
                        0.000,
                                  0.000 >
define b4 < -1.000,
                        0.000,
                                  0.000 >
define b5 <
              0.000,
                      -1.000,
                                  0.000 >
define r1 0.96
define r2 0.96
define r3 0.98
define r4 0.99
define r5 1.00
define r6 1.01
define r7 1.02
define r8 1.03
define r9 1.04
define range 0.25
if (frame < total_frames/2)</pre>
   define strength range*10^(frame/total_frames)+0.7
else
    define strength range*10^((total_frames_frame)/total_frames)+0.7
define xrot frame*360/total.frames
define threshold \
// fire 4
object {
   blob threshold:
      strength, r1,b1,
      strength, r1,b2,
      strength, r1,b3,
      strength, r1,b4,
      strength, r1,b5
   root solver Ferrari
   u_steps 20
   v_steps 20
   reflective_coral
   rotate <xrot, xrot, 0>
   translate <-2.500,-1.000, 1.500 >
   }
object {
   blob threshold:
      strength, r2,b1,
      strength, r2,b2,
      strength, r2,b3,
      strength, r2,b4,
      strength, r2,b5
   root_solver Ferrari
```

```
u_steps 20
   v_steps 20
   reflective_coral
   rotate <xrot,0, 0>
   translate < 0.000,-1.000, 1.500 >
object {
   blob threshold:
      strength, r3,b1,
      strength, r3,b2,
      strength, r3,b3,
      strength, r3,b4,
      strength, r3,b5
   root solver Ferrari
   u_steps 20
   v_steps 20
   reflective_coral
   rotate <xrot,-xrot, 0>
   translate < 2.500,-1.000, 1.500 >
   }
object {
   blob threshold:
      strength, r4,b1,
      strength, r4,b2,
      strength, r4,b3,
      strength, r4,b4,
      strength, r4,b5
   root_solver Ferrari
   u_steps 20
   v_steps 20
   reflective_coral
   rotate <-xrot,xrot, 0>
   translate <-2.500, 0.000,-0.250 >
object {
   blob threshold:
      strength, r5,b1,
      strength, r5,b2,
      strength, r5,b3,
      strength, r5,b4,
      strength, r5,b5
   root_solver Ferrari
   u_steps 20
   v_steps 20
   reflective_coral
   rotate <-xrot,0, 0>
   translate < 0.000, 0.000,-0.250 >
   }
object {
   blob threshold:
```

continued on next page

```
continued from previous page
      strength, r6,b1,
      strength, r6,b2,
      strength, r6,b3,
      strength, r6,b4,
      strength, r6,b5
   root_solver Ferrari
   u_steps 20
   v_steps 20
   reflective_coral
   rotate <-xrot,-xrot, 0>
   translate < 2.500, 0.000,-0.250 >
   }
object {
   blob threshold:
      strength, r7,b1,
      strength, r7,b2,
      strength, r7,b3,
      strength, r7,b4,
      strength, r7,b5
   root_solver Ferrari
   u_steps 20
   v_steps 20
   reflective_coral
   rotate <xrot, xrot, 0>
   translate <-2.500, 1.000,-2.000 >
   }
object {
   blob threshold:
      strength, r8,b1,
      strength, r8,b2,
      strength, r8,b3,
      strength, r8,b4,
      strength, r8,b5
   root_solver Ferrari
   u_steps 20
   v_steps 20
   reflective_coral
   rotate <xrot,0, 0>
   translate < 0.000, 1.000,-2.000 >
object {
   blob threshold:
      strength, r9,b1,
      strength, r9,b2,
      strength, r9,b3,
      strength, r9,b4,
      strength, r9,b5
   root_solver Ferrari
   u_steps 20
   v_steps 20
   reflective_coral
   rotate <xrot,-xrot, 0>
   translate < 2.500, 1.000,-2.000 >
   }
```

How It Works

This is an example of an animation created to test out the control variables for what was a new graphics primitive at the time this code was written. The objects' reflections surface rotate making this interesting animation stand out (Figure 7-10). This also serves as an example of how to explore aspects of ray tracing code when you're interested in seeing how parameters affect the formations of surfaces.

We select nine values for the radii of the blob elements:

```
define r1 0.96
define r2 0.96
define r3 0.98
define r4 0.99
define r5 1.00
define r6 1.01
define r7 1.02
define r8 1.03
define r9 1.04
```

then vary the strength parameter during the animation and watch the results:

```
if (frame < total_frames/2)
  define strength 0.00075*10^(frame/80)
else
  define strength 0.00075*10^((total_frames-frame)/80)</pre>
```

Granted, this is over a fairly narrow range, and it resulted from explorations with wider ranges when the old Polyray bug was active.

Comments

This type of exploration is particularly useful when hunting for new textures, since you can cover a lot of ground automatically, programming your animations to search out and explore what the various parameters control.

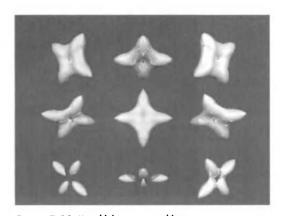


Figure 7-10 Nine blob crosses tumbling in unison



7.4 How do I...

Resort to udder nonsense?

You'll find the code for this in: PLY\CHAPTER7\UDDER

Problem

John Hammerton and I came up with some interesting blobs that, for lack of a better description, resemble cows' udders. He placed two next to each other, one with six teats and one with eight, and counter-rotated them. An obvious next step in this process is to pong the two sets. Pong means to go back and forth between two states, much like the ball in a ping-pong game. We start with six on the left, eight on the right, pull both sets in, then extrude them so that the left side has eight and the right side has six (Figure 7-11). We can go back and forth between these two forms in an endless, mindless loop.

Technique

Pressing the fast-forward button here, having already generated this animation and watched it, the most disappointing aspect is that after all the effort spent to pong the teats, it's hard to tell that it's happening. There are not enough clues. What we need to do is also pong the colors.

One udder is bronzy, the other one is chrome. We generate two variables based on the animation index, so that when one of them is is zero, the other of them is one, and vice versa, and together they always add up to 1. We use

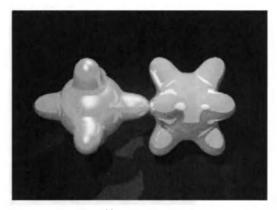


Figure 7-11 Pong udders

these factors in the texture block to vary all the controlling factors we find, and manage to pong the textures as well. Then the effect we're after is noticeable:

```
// uddernon.pi - Udder Nonsense
start frame O
end_frame 44
total_frames 45
define index frame/total_frames
outfile "udd"
define spin 90*index
include "\PLY\COLORS.INC"
viewpoint {
   from <0,8,-8>
   at <0,0,0>
   up <0,1,0>
   angle 30
   resolution 320,200
   aspect 1.43
background SkyBlue*0.75
spot_light 0.6 * white, <-2,10,-5>,<0,0,0>,3,5,20
spot_light 0.6 * white, < 2,10, 5>,<0,0,0>,3,5,20
define a1 < 1.73205, 0, 0>
define a2 < 0, 1.73205, 0>
define a3 < 0, 0, 1.73205>
define a4 <-1.73205, 0, 0>
define a5 < 0,-1.73205, 0>
define a6 < 0, 0, -1.73205>
define a7 < 0, 0, 0>
define a8 < 0, 0, 0 >
define b1 < 1, 1, 1>
define b2 < 1, 1, -1 >
define b3 < 1,-1, 1>
define b4 < 1,-1,-1>
define b5 <-1, 1, 1>
define b6 <-1, 1,-1>
define b7 <-1,-1, 1>
define b8 <-1,-1,-1>
define pi 3.14159
define sense cos(2*pi*index)
// conjugates
define t1 ((sense+1)/2)
define t2 (1-t1)
```

```
continued from previous page
define bronzy
texture {
surface {
   ambient <1.0,0.5*t1+1.0*t2,0.0*t1+1.0*t2>,0.2*t1 + 0.1*t2
   diffuse <1.0,0.5*t1+1.0*t2,0.0*t1+1.0*t2>,0.6*t1 + 0.2*t2
   specular white, 0.8*t1 + 0.0*t2
   reflection <1.0,0.5*t1+1.0*t2,0.0*t1+1.0*t2>, 1
   color <1.0*t1+0.1*t2,0.5*t1+0.1*t2,0.0*t1+0.1*t2>
   }
}
define chrome
texture {
   surface {
      ambient <1.0,1.0*t1+0.5*t2,1.0*t1+0.0*t2>, 0.1*t1 + 0.2*t2
      diffuse <1.0,1.0*t1+0.5*t2,1.0*t1+0.0*t2>, 0.2*t1 + 0.6*t2
      specular white, 0.0*t1+0.8*t2
      reflection <1.0,1.0*t1+0.5*t2,1.0*t1+0.0*t2>, 1
      color <0.1*t1+1.0*t2,0.1*t1+0.5*t2,0.1*t1+0.0*t2>
   }
if (sense > 0) {
 define c1 a1*sense
  define c2 a2*sense
  define c3 a3*sense
  define c4 a4*sense
  define c5 a5*sense
  define c6 a6*sense
  define c7 a7*sense
 define c8 a8*sense
  define d1 b1*sense
  define d2 b2*sense
  define d3 b3*sense
  define d4 b4*sense
  define d5 b5*sense
  define d6 b6*sense
  define d7 b7*sense
 define d8 b8*sense
else {
 define c1 b1*sense
 define c2 b2*sense
 define c3 b3*sense
 define c4 b4*sense
  define c5 b5*sense
  define c6 b6*sense
  define c7 b7*sense
 define c8 b8*sense
 define d1 a1*sense
  define d2 a2*sense
```

```
define d3 a3*sense
  define d4 a4*sense
  define d5 a5*sense
  define d6 a6*sense
  define d7 a7*sense
  define d8 a8*sense
}
object {
  blob 3.5:
    sphere <0,0,0>, 6,2.2,
    sphere c1, 3, 1,
    sphere c2, 3, 1,
    sphere c3, 3, 1,
    sphere c4, 3, 1,
    sphere c5, 3, 1,
    sphere c6, 3, 1,
    sphere c7, 3, 1,
    sphere c8, 3, 1
    rotate <0, spin, 0>
    translate <-1.5,0,0>
    bronzy
)
object {
  blob 3.5:
    sphere <0,0,0>, 6,2.2,
    sphere d1, 3, 1,
    sphere d2, 3, 1,
    sphere d3, 3, 1,
    sphere d4, 3, 1,
    sphere d5, 3, 1,
    sphere d6, 3, 1,
    sphere d7, 3, 1,
    sphere d8, 3, 1
    rotate <0, -spin, 0>
    translate <1.5,0,0>
    chrome
}
object { disc <0,-2,0>,<0,1,0>,15 matte_blue }
```

How It Works

We'll start with the udder pong. We define two sets of eight vectors each, a1-a8 for the six-udder object (the last two vectors are zero) and b1-b8 for the eight-udder object. We control the pong with a simple cosine wave called sense. Two factors t1 and t2 are our conjugates. The texture blocks are unusually busy, since we use t1 and t2 to set the levels of each item that controls how a surface is rendered:

```
define bronzy
texture {
surface {
   ambient <1.0,0.5*t1+1.0*t2,0.0*t1+1.0*t2>,0.2*t1 + 0.1*t2
   diffuse <1.0,0.5*t1+1.0*t2,0.0*t1+1.0*t2>,0.6*t1 + 0.2*t2
   specular white, 0.8*t1 + 0.0*t2
   reflection <1.0,0.5*t1+1.0*t2,0.0*t1+1.0*t2>, 1
   color <1.0*t1+0.1*t2,0.5*t1+0.1*t2,0.0*t1+0.1*t2>
   }
}
```

We use a conditional that, based on the sign of the value of *sense*, places the teats on the left or the right blob object, and multiplies the control vectors so that the effect grows smoothly out of one, receeds, then grows smoothly out of the other. We assign the textures to these blobs and set them spinning.



7.5 How do I...

Fly through a soothing fluid tunnel?

You'll find the code for this in: PLY\CHAPTER7\WATRTUNL

Problem

Aligning and smoothing out the intersections of cylinders for the creation of fly-through tunnels can be difficult. In contrast, cylindrical blobs make great intersecting tunnels. Rather than having to deal explicitly with the intersection point, blob tunnels open up naturally into each other.

Technique

In WATRTUNL.PI, a fluid texture is applied to a series of intersecting blob tunnels. The camera moves through the tunnel to make a soothing animation. It does, however, take a very long time to render, due to all the reflections:

```
// WATRTUNL.PI - Drifting Through a Reflective Blob Tunnel
start_frame 0
end_frame 119
total_frames 120
outfile watr
define index 360/total_frames
define norm frame/total_frames
```

```
viewpoint {
   from <0, 0.01,-6+12*norm>
         <0, 1, 0>
   up
         <0, 0, 10>
   аt
   angle 25
   resolution 320,200
   aspect 1.43
light <1,1,0>,< 0, 0, -6+12*norm>
light <-5, 0, 0>
light < 5, 0, 0>
background midnightblue
define pi 3.14159
define rad pi/180
include "..\colors.inc"
define blue_ripple
texture {
  noise surface {
      color <0.4, 0.4, 1.0>
      normal 2
      frequency 16
      phase -frame*index*rad
     bump_scale 2
     ambient 0.3
     diffuse 0.4
      specular yellow, 0.7
  // reflection 0.5
     microfacet Reitz 10
  rotate <90,0,0>
object {
  blob 0.5:
// the side chambers
      cylinder <-2, 0, -6>, < 2, 0, -6>, 1, 0.5,
      cylinder <-2, 0, -4>, < 2, 0, -4>, 1, 0.5,
      cylinder <-2, 0, -2>, < 2, 0, -2>, 1, 0.5,
     cylinder <-2, 0, 0>, < 2, 0, 0>, 1, 0.5,
      cylinder <-2, 0, 2>, < 2, 0, 2>, 1, 0.5,
      cylinder <-2, 0, 4>, < 2, 0, 4>, 1, 0.5,
      cylinder <-2, 0, 6>, < 2, 0, 6>, 1, 0.5,
// the tube
      cylinder < 0, 0, -6>, < 0, 0, 6>, 1, 0.5
     blue_ripple
  }
```



Figure 7-12 Midpoint view inside our tunnel

How It Works

This is a simple but beautiful animation. A single blob tunnel is intersected by a series of other tunnels at right angles. The camera viewpoint moves down the tunnel, accompanied by a light. The phase of the surface normals for the rippling texture changes as we move down the tube. We proceed from one end of the tunnel to the other. A midpoint view is shown in Figure 7-12.

Comments

The addition of other elements drifting past inside the tunnel would enhance the sense of motion. However, this animation took over a week to render as it stands. The variable "reflection" in the *blue_ripple* texture definition can be commented out to speed up the rendering with only a minor impact on the image.



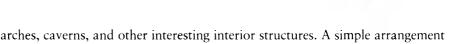
7.6 How do I...

Fly through tunnels on an asteroid?

You'll find the code for this in: PLY\CHAPTER7\ASTEROID

Problem

One of the real challenges of computer animation is the coordination of a flight path inside an object at close quarters. One example of this is a fight through the interior of a blob. As was mentioned in Section 3.9, it was quite a revelation to discover that not only are blobs hollow, they contain tunnels,



blob cavern. It's a real challenge to design a spline path that threads these caverns without smashing through the walls.

The other chief difficulty in flying around randomly inside an object is the dreaded "up" vector. The definition of the camera's viewpoint is fine every place except where the camera is pointing either directly up or directly down. At these two points, the idea of "up" becomes undefined, so the camera no longer maintains a stable orientation. It can flip suddenly upside down with the slightest motion on either side of where "up" is, resulting in your scene doing an abrupt 180° turn. The asteroid flythrough could deal with this problem by continuously redefining the position of up so that it always points to the center of the asteroid, but this numbs the sense of up and down (well, actually, it kills it entirely). Instead, we first run a small test animation, determine at what frame numbers the flips occurs, and then correct the camera orientation to obtain consistent camera viewpoint.

of metaballs in a regular cubical structure generates an amazing three-level

Technique

The most demanding part of this animation was threading the flight path through the tunnels. A cutaway view reveals where the bottoms of the tunnels are, and given the regular placement of metaballs, leads to a collection of spline control points that steer us through the caverns and lets us exit without a scratch.

The first step was to render the asteroid we created with the outer layers removed to give us an idea where the floors of the tunnels were. Once you realize that the metaballs are placed on the corners and along the edges of a cube, it's not surprising to find that the tunnel also lines up with these features. A collection of data points based on the edges and corners of the cube line up with the insides of the tunnels. It then becomes a matter of threading these points in the proper sequence with a spline line to smoothly go from one clear area to the next, eventually moving down each tunnel. The initial guess is shown in Table 7-1:

X	Y	Z
0	0	2
0	0	1
1	1	1
0	1	0

continued on next page

continued from previous page

X	Y	Z
1	1	-1
1	0	0
1	-1	-1
0	0	-1
-1	-1	-1
-1	0	0
-1	1	-1
0	1	0
-1	1	1
0	0	1
-1	-1	1
0	-1	0
1	-1	1
1	0	0
2	0	0

Table 7-1 Initial placement of spline control points around the asteroid

This had some hard cornering going from the corner points (such as <1,1,1>) to the axis points (like <1,0,0>) since the corner points were 73 percent farther out. The collection was softened by moving the outer points in a bit. These new control points are shown in Table 7-2:

X	Y	Z	
0.0	0.0	16.0	
0.0	0.0	8.0	
0.0	0.0	4.0	
0.0	0.0	2.0	
0.0	0.0	1.5	
0.1	0.0	1.0	
0.0	0.0	0.5	
0.5	0.5	0.5	
0.0	0.75	0.0	
0.5	0.5	-0.5	

X	Y	Z	
0.75	0.0	0.0	
0.5	-0.5	-0.5	
0.0	0.0	-0.75	
-0.5	-0.5	-0.5	
-0.75	0.0	0.0	
-0.5	0.5	-0.5	
0.0	0.75	0.0	
-0.5	0.5	0.5	
0.0	0.0	0.75	
-0.5	-0.5	0.5	
0.0	-0.75	0.0	
0.5	-0.5	0.5	
0.5	0.0	0.0	
1.0	0.0	0.0	
2.0	0.0	0.0	
4.0	0.0	0.0	
8.0	0.0	0.0	
16.0	0.0	0.0	

Table 7-2 Adjusted spline control points

We've added some additional points to decelerate towards and accelerate away from the asteroid. A rendering of the control points and the resulting spline path is shown in Figure 7-13.

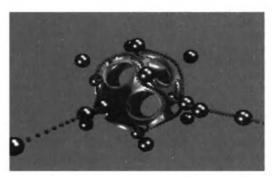


Figure 7-13 Cutaway view of the blob asteroid with the spline path overlaid

The following code takes our control points and creates the spline path. It writes the camera locations (*frx*, *fry*, and *frz*) and what the camera is pointed at (*atx*, *aty*, and *atz*) to a batch file that changes those values in an include file called ANIM, then calls Polyray, which uses a master file ASTEROID.PI with the camera file included to render the scene.

```
' PATH3.BAS - Asteroid Spline Path Creation Program
  "Magic Spline Fitting Code" translated from the C spline code in
  INTERACTIVE 3D COMPUTER GRAPHICS by L. Ammeraal,
' Chichester: John Wiley & Sons. (1988)
DECLARE SUB rotate (a!, b!, c!)
TYPE vector
  x AS SINGLE
 y AS SINGLE
  z AS SINGLE
END TYPE
SCREEN 12
WINDOW (-3.2, -2.4)-(3.2, 2.4)
LINE (-1, -1)-(1, 1), B
' left to right tunnels
s = 8
b = 1
LINE (-4, -4)-(4, 4), B
' The individual control points.
DATA 0.0,
           0.0, 16.0
DATA 0.0,
           0.0,
                  8.0
                   4.0
DATA 0.0,
           0.0,
DATA 0.0,
           0.0,
                   2.0
DATA 0.0,
                   1.5
           0.0,
           0.0,
DATA 0.1,
                   1.0
DATA 0.0,
           0.0,
                   0.5
DATA 0.5,
           0.5,
                   0.5
DATA 0.0,
           0.75,
                  0.0
DATA 0.5,
           0.5, -0.5
DATA 0.75, 0.0,
                  0.0
DATA 0.5, -0.5,
                 -0.5
DATA 0.0,
           0.0,
                 -0.75
DATA-0.5, -0.5,
                 -0.5
DATA-0.75, 0.0,
                   0.0
DATA-0.5,
           0.5, -0.5
DATA O.O,
           0.75, 0.0
           0.5,
                  0.5
DATA-0.5,
DATA 0.0,
           0.0,
                  0.75
```

```
DATA-0.5, -0.5,
                   0.5
DATA 0.0, -0.75,
                   0.0
DATA 0.5, -0.5,
                   0.5
DATA 0.5,
          0.0,
                   0.0
DATA 1.0,
          0.0,
                   0.0
DATA 2.0,
          0.0,
                   0.0
DATA 4.0,
          0.0,
                   0.0
DATA 8.0,
          0.0,
                   0.0
DATA 16.0, 0.0,
                   0.0
DATA 32.0, 0.0,
                   0.0
m = 29
n = 10
scale = .9
count = 500
ahead = 5
DIM x(m), y(m), z(m)
DIM cam(count + ahead) AS vector
FOR i = 1 TO m
  READ x(i), y(i), z(i)
NEXT i
k = 1
m = 28
OPEN "check" FOR OUTPUT AS #2
FOR i = 2 TO m - 1
       ' MAGIC SPLINE FITTING CODE
       xA = x(i - 1): xB = x(i): xC = x(i + 1): xD = x(i + 2)
       yA = y(i - 1): yB = y(i): yC = y(i + 1): yD = y(i + 2)
       zA = z(i - 1): zB = z(i): zC = z(i + 1): zD = z(i + 2)
       a3 = (-xA + 3! * (xB - xC) + xD) / 6!
       a2 = (xA - 2! * xB + xC) / 2!
       a1 = (xC - xA) / 2!
       a0 = (xA + 4! * xB + xC) / 6!
       b3 = (-yA + 3! * (yB - yC) + yD) / 6!
       b2 = (yA - 2! * yB + yC) / 2!
       b1 = (yC - yA) / 2!
       b0 = (yA + 4! * yB + yC) / 6!
       c3 = (-zA + 3! * (zB - zC) + zD) / 6!
       c2 = (zA - 2! * zB + zC) / 2!
       c1 = (zC - zA) / 2!
       c0 = (zA + 4! * zB + zC) / 6!
       'PRINT USING "< ##.####, ##.####, ##.### >"; a3; b3; c3
```

20

continued on next page

```
continued from previous page
       'PRINT USING "< ##.####, ##.####, ##.### >"; a2; b2; c2
       'PRINT USING "< ##.####, ##.####, ##.### >"; a1; b1; c1
       'PRINT USING "< ##.####, ##.#### >"; aO; bO; cO
       'PRINT
       w = 2
       FOR j = first TO n
              t = j / n
              cam(k).x = scale * (((a3 * t + a2) * t + a1) * t + a0)
               cam(k).y = scale * (((b3 * t + b2) * t + b1) * t + b0)
              cam(k).z = scale * (((c3 * t + c2) * t + c1) * t + c0)
              PRINT #2, USING "##.##### "; cam(k).x; cam(k).y; cam(k).z
              k = k + 1
              IF j = first THEN
                      PSET (x, z), y * 20 + 8
                      CIRCLE (x, z), .05, 2
              ELSE
                      PSET (x, z), y * 20 + 8
              END IF
       NEXT j
       first = 1
NEXT i
CLOSE #2
OPEN "orbiter.bat" FOR OUTPUT AS #1
k = 1
frame = 0
DO WHILE cam(k).x < 4
  IF cam(k).z < 4 THEN
  frame = frame + 1
  count$ = RIGHT$("0000" + LTRIM$(STR$(frame)), 4)
  PRINT #1, USING "echo define frx ##.##### >anim"; cam(k).x
  PRINT #1, USING "echo define fry ##.##### >>anim"; cam(k).y
  PRINT #1, USING "echo define frz ##.#### >>anim"; cam(k).z
  up = 1
  dimn = 1.5
  norm1 = (frame - 20) / 40
  norm2 = (frame - 190 - 20) / 40
' manually flip "up" to prevent scene reversal
  IF frame > 78 AND frame < 119 THEN up = -1
  IF frame > 158 THEN up = -1
  inc = 0
  IF frame < 20 THEN inc = 1
  IF frame > 190 THEN inc = 2
```

```
PRINT #1, USING "echo define atx ##.#### >>anim"; cam(k + ahead).x
  PRINT #1, USING "echo define aty ##.#### >>anim"; cam(k + ahead).y
  PRINT #1, USING "echo define atz ##.##### >>anim"; cam(k + ahead).z
  PRINT #1, USING "echo define upx ##.##### >>anim"; O!
  PRINT #1, USING "echo define upy ##.##### >>anim"; up
  PRINT #1, USING "echo define upz ##.##### >>anim"; O!
  PRINT #1, USING "echo define inc # >>anim"; inc
       PRINT #1, "\ply\polyray astr2.pi -o astr"; count$; ".tga "
       PRINT #1,
       a = cam(k).x
       b = cam(k).y
       c = cam(k).z
       CALL rotate(a, b, c)
       d = cam(k + ahead).x
       e = cam(k + ahead).y
       f = cam(k + ahead).z
       CALL rotate(d, e, f)
       ' show it
       LINE (a, c)-(d, f), 1 + co MOD 14
       co = co + 1
    END IF
    k = k + 1
L00P
CLOSE #1
SUB rotate (a, b, c)
pi = 3.14159
rad = pi / 180
'rotate
        xrotate = 30
        yrotate = 45
        zrotate = 0
        x0 = a
        y0 = b
        z0 = c
        x1 = x0
        y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
        z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)
        x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
        y2 = y1
                                                                  continued on next page
```

```
z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)

x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)

y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)

z3 = z2

a = x3
b = y3
c = z3
```

END SUB

How It Works

This program generates the batch file ORBITER.BAT, which is shown below:

```
echo define frx 0.000000 >anim
echo define fry 0.000000 >>anim
echo define frz 3.900000 >>anim
echo define atx 0.000000 >>anim
echo define aty 0.000000 >>anim
echo define atz 2.765625 >>anim
echo define upx 0.000000 >>anim
echo define upy 1.000000 >>anim
echo define upz 0.000000 >>anim
echo define inc 1 >>anim
\ply\polyray astr2.pi -o astr0001.tga
echo define frx 0.000000 >anim
echo define fry 0.000000 >>anim
echo define frz 3.638925 >>anim
echo define atx 0.000000 >>anim
echo define aty 0.000000 >>anim
echo define atz 2.587800 >>anim
echo define upx 0.000000 >>anim
echo define upy 1.000000 >>anim
echo define upz 0.000000 >>anim
echo define inc 1 >>anim
\ply\polyray astr2.pi -o astr0002.tga
```

This batch file writes a file called ANIM, which defines items like the camera location, what the camera is pointed at, and the orientation of the up vector. The Polyray file ASTR2.PI contains lights and our asteroid. It looks like this:

```
// ASTR2.PI
// interior blob reflection
include "\ply\colors.inc"
include "anim"
```

```
viewpoint {
   from < frx, fry, frz >
       < atx, aty, atz >
       < upx, upy, upz >
   up
   angle 65
   resolution 320,200
   aspect 1.433
   }
background SkyBlue
define dim 0.4
light <dim,dim,dim>,<frx+1,fry+0,frz+0>
light <dim,dim,dim>,<frx+0,fry+1,frz+0>
light <dim,dim,dim>,<frx+0,fry+0,frz+1>
light <dim,dim>,<frx-1,fry+0,frz+0>
light <dim,dim>,<frx+0,fry-1,frz+0>
light <dim,dim>,<frx+0,fry+0,frz-1>
light <0.3,0.15,0.6>, <frx,fry,frz>
define asteroid
object {
   blob 6.6:
      7, 3.0, < 0, 0, 0 >,
      3, 1.0,< 1, 1, 1 >,
      3, 1.0, < -1, 1,
                       1 >.
      3, 1.0, < 1, -1,
      3, 1.0, < -1, -1, 1 >
     3, 1.0,< 1, 1, -1 >,
      3, 1.0, < -1, 1, -1 >
      3, 1.0,< 1, -1, -1 >,
      3, 1.0,< -1, -1, -1 >,
      3, 1.0,< 0, 1, 1 >,
      3, 1.0, < 0, 1, -1 >
      3, 1.0, < 0, -1,
                      1 >,
      3, 1.0, < 0, -1, -1 >
      3, 1.0, < 1, 0, 1 >,
      3, 1.0, < 1, 0, -1 >
      3, 1.0, < -1, 0, 1 >
      3, 1.0, < -1, 0, -1 >
      3, 1.0,< 1, 1, 0 >,
      3, 1.0, < 1, -1, 0 >
      3, 1.0, < -1, 1, 0 >
      3, 1.0, < -1, -1,
   root_solver Sturm
   u_steps 20
   v_steps 20
   shiny_coral
```

continued on next page

CHAPTER SEVEN

```
continued from previous page
define s 8
define collection
object {
   asteroid {translate <-s,-s,-s>}
+ asteroid {translate <-s,-s, 0>}
+ asteroid {translate <-s,-s, s>}
+ asteroid {translate <-s, 0,-s>}
+ asteroid {translate <-s, 0, 0>}
+ asteroid {translate <-s, 0, s>}
+ asteroid {translate <-s, s,-s>}
+ asteroid {translate <-s, s, 0>}
+ asteroid {translate <-s, s, s>}
+ asteroid {translate < 0,-s,-s>}
+ asteroid {translate < 0,-s, 0>}
+ asteroid {translate < 0,-s, s>}
+ asteroid {translate < 0, 0,-s>}
+ asteroid {translate < 0, 0, 0>}
+ asteroid {translate < 0, 0, s>}
+ asteroid {translate < 0, s,-s>}
+ asteroid {translate < 0, s, 0>}
 + asteroid {translate < 0, s, s>}
+ asteroid {translate < s,-s,-s>}
+ asteroid {translate < s,-s, 0>}
+ asteroid {translate < s,-s, s>}
+ asteroid {translate < s, 0,-s>}
+ asteroid {translate < s, 0, 0>}
+ asteroid {translate < s, 0, s>}
+ asteroid {translate < s, s,-s>}
+ asteroid {translate < s, s, 0>}
+ asteroid {translate < s, s, s>}
// main one
asteroid
if(inc==1) {collection {translate <0, 0, -2*s> }}
if(inc==2) {collection {translate < 2*s, 0, 0> }}
```

Rather than making a single pass through this asteroid, a loop point was made by flying in through one side and exiting out the other, and by having a field of 27 other asteroids called *collection* appear at the beginning and the end of the flight, controlled by the variable *inc*. This makes it appear that you're in an asteroid field, but once you are inside the caverns there's little point in having 27 extra blobs slow down the rendering. Both *collections* are made to line up exactly, so that the first frame aligns with the last. This makes it appear as if you were entering the same asteroid over and over. To maintain similar lighting at both ends, all light sources track the motion of the camera and end up exactly where they were when they began, with respect to the asteroid we're about to enter. There's still a flash as some of the light sources emerge from behind some blobs, but it's minor.

The asteroids in the array are all separated by eight units. Once you're outside the asteroid, you accelerate away from one towards the next one. Conditional code generates frames that start at the midpoint between two blob asteroids and end at roughly the same position.

Views of the asteroid field and the inside of the tunnel are shown in Figures 7-14 and 7-15.

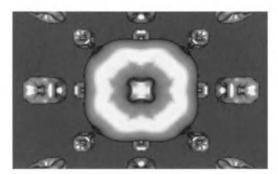


Figure 7-14 The asteroid field

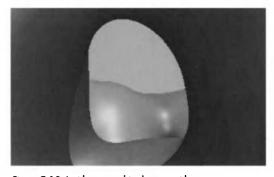


Figure 7-15 Inside a tunnel in the asteroid

CHAPTER



8

RECURSION

any patterns in nature derive their pleasant symmetry as well as their low level details to some very simple construction rules applied across a wide range of scales in an image. For example, an object is defined as an assemblage of smaller objects using some rule. Those objects are themselves defined using this same rule, and the process continues down the resolution scale until the details become too small to see or the computer gags on the number of objects it's asked to generate. Calling image definition code from within a layer that uses that same code is known as recursion, and it automates the process of defining finely detailed objects.

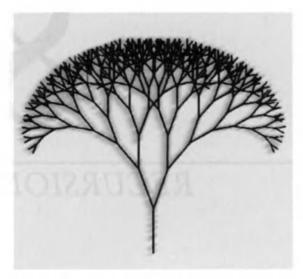


Figure 8-1 A recursive tree

Take a look at the L-system fractal tree in Figure 8-1. The entire tree was defined with just two parameters: the angle of the branches and the rate at which the branches get smaller. The computer just iterates the code to get the complex shape of the tree.

To create such recursive images with Polyray, we can either use external files and batch animation, or use nested defines in the ray tracer, such as:

```
define layer1
object {
    object { sphere ... }
    + object { cylinder... }
    + ...
}

define layer2
object {
    object {layer1 scale <...> translate <...> }
    + ...
}

define layer3
object {
    object {layer2 scale <...> translate <...> }
    + ...
}
```



We'll play with two kinds of recursion: physical recursion that generate complex objects from simple definitions, and motion recursion, where the movements of smaller pieces of an object are synchronized with the movements of larger pieces. Physical recursion results in an object being self-similar; in other words, parts of it resemble other parts, only on different scales. Motion recursion may involve something like taking the seed element for a recursive object and rotating it, setting all the smaller pieces into coordinated motions that mimic the larger motion. This is very easy in Polyray, since motion can be included as part of an object when it's defined, and the effects can be bizarre.

More so than any other chapter, this one is extremely open-ended. The examples just provide the starting point for limitless exploration into recursion and self-similar animations.



ഁ 8.1 How do I...

Grow an L-system fan?

You'll find the code for this in: PLY\CHAPTER8\FAN

Problem

Figure 8-1 shows a detailed object that was generated not by the tedious process of hand crafting each little item, but by giving instructions to a computer that takes simple specifications and does the work for us. We'll use recursion to generate a branched fractal tree, and animate the variables to illustrate their effects.

Technique

A recursive program is written in QuickBasic, modeled on code found in an excellent book by Roger Stevens, *Fractal Programming in C* (M&T Books 1989). It has three parts. A main routine defines control variables and draws a seed element. A recursive subroutine is used to draw the branches. At each new branch, we need to know what the angle was from the last branch, and build the next one with respect to it. New branches form to the left and right of the previous ones. The main difference between recursion in QuickBasic and recursion in C is that in C, just the value of variables get passed to the subroutine, not the variables themselves. QuickBasic defaults to passing the actual variables, which are then modified while they're in the subroutine. This makes it necessary for us to save the variables before we enter the subroutine, and restore them on returning.



Figure 8-2 Frame 15 from the Fan animation

Steps

The following code (FAN.BAS) generates all the necessary programs to generate a recursive fan that's illustrated in Figure 8-2.

' FAN.BAS

```
DECLARE SUB Branch (x1, y1, height, level)
DECLARE FUNCTION Ang (x1, y1, x2, y2)
COMMON SHARED rad, theta, scale, angle, k, twig
SCREEN 12
WINDOW (-64, -10)-(64, 86)
       angle = 10
       twig = 10
frame = 0
OPEN "anim.bat" FOR OUTPUT AS #2
FOR sc = .6 \text{ TO } .86 \text{ STEP } .01
  frame = frame + 1
  inc$ = "fan" + RIGHT$("00" + LTRIM$(STR$(frame)), 2) + ".inc"
  tga$ = "fan" + RIGHT$("00" + LTRIM$(STR$(frame)), 2) + ".tga"
  PRINT #2, "echo include "; CHR$(34); inc$; CHR$(34); " > anim.inc"
  PRINT #2, "\ply\polyray fan.pi -o "; tga$
  OPEN inc$ FOR OUTPUT AS #1
       scale = sc ^ .5
       rad = 3.14159 / 180
       height = 10
       twig = twig - .2
```

```
angle = angle + .5
       level = 8
       x1 = 0
       y1 = 0
       x2 = 0
       y2 = y1 + height
       LINE (x1, y1)-(x2, y2)
       PRINT #1, "object {"
       PRINT #1, USING " object {cone ←
<###.###,###.###,###.###>,###.###,<###.###.###.###.###.###.###</pre>
matte_white \)"; x1, y1, 0, height * scale / twig, x2, y2, 0, height * scale \Leftarrow
* scale / twig
      ' left-hand sides
       theta = Ang(x1, y1, x2, y2)
       theta = theta + angle
       levelsave = level
       CALL Branch(x2, y2, scale * height, level)
       level = levelsave
      ' right-hand sides
       theta = Ang(x1, y1, x2, y2)
       theta = theta - angle
       CALL Branch(x2, y2, scale * height, level)
       FOR w = 1 TO 100000: NEXT w
       CLS
  PRINT #1, "}"
  CLOSE #1
NEXT sc
CLOSE #2
FUNCTION Ang (x1, y1, x2, y2)
' returns the angle of a vector with respect to the x-axis
       IF ((x2 - x1) = 0) THEN
               IF (y2 > y1) THEN
                      theta = 90
               ELSE
                      theta = 270
         END IF
       ELSE
               theta = ATN((y2 - y1) / (x2 - x1)) / rad
  END IF
  IF (x1 > x2) THEN theta = theta + 180
  Ang = theta
END FUNCTION
                                                                   continued on next page
SUB Branch (x1, y1, height, level)
```

```
continued from previous page
      x = x1
      y = y1
      x = x + height * COS(theta * rad)
      y = y + height * SIN(theta * rad)
      x2 = x
      y2 = y
      level = level - 1
      LINE (x1, y1)-(x2, y2), 15
      PRINT #1, USING " + object {cone ←
matte_white \}"; x1, y1, 0, height * scale / twig, x2, y2, 0, height * scale \Leftarrow
* scale / twig
      IF (level > 0) THEN
            ' left-hand sides
              theta = Ang(x1, y1, x2, y2)
             theta = theta + angle
             levelsave = level
             CALL Branch(x2, y2, scale * height, level)
             level = levelsave
            ' right-hand sides
             theta = Ang(x1, y1, x2, y2)
              theta = theta - angle
              levelsave = level
              CALL Branch(x2, y2, scale * height, level)
              level = levelsave
  END IF
END SUB
```

How It Works

The program draws the initial trunk, then calls a function that determines the angle of that segent with respect to the x axis. It adds *angle* to it, and goes to a subroutine that generates a branch at that angle, but not before it saves the current level:

```
theta = Ang(x1, y1, x2, y2)
theta = theta + angle
levelsave = level
CALL Branch(x2, y2, scale * height, level)
level = levelsave
```

The *Branch* subroutine calculates a vector at the appropriate angle and length and adds it to the current top of the tree. It decrements the level and, if it's not at the last level, it calls itself. All the left-hand branch sides are completed first. It then goes back and completes the right hand sides, starting at the tips and working its way back to the trunk.

We index the scale so that the fan grows a little bit longer each frame. We also index the diameter of the branches so the tree gets thicker as it grows downward. We use cones rather than cylinders to define the branches, so that as the twigs get smaller and smaller, they still match at the junctions.

A short Polyray data file uses this information to visualize our tree.

```
include "\ply\colors.inc"

viewpoint {
    from <0,75,-125>
    at <0,32.5,0>
    up <0,1,0>
    angle 30
    resolution 320,200
    aspect 1.433
    }
background SkyBlue

object {disk <0,0,0>, <0,1,0>, 1500 matte_blue}

spot_light <2,1,.5>, <0,500,-500>,<0,0,0>,3,5,20
include "anim.inc"
```

The animation uses external include files containing 511 cones each, numbered from FAN01.INC to FAN27.INC. We use the normal animation batch file that writes the desired include file for that frame and calls Polyray, giving it sequential output files to write to.

```
echo include "c:\qb\fan01.inc" > anim.inc
\ply\polyray fan.pi -o fan01.tga
echo include "c:\qb\fan02.inc" > anim.inc
\ply\polyray fan.pi -o fan02.tga
...
```

The output grows from a slender form to a flat fan that fills the screen. Frame 15 is shown in Figure 8-2.



8.2 How do I...

Generate recursively layered objects?

You'll find the code for this in: PLY\CHAPTER8\ITERBOX

Problem

Objects comprised of thousands of individual pieces can be tedious to design. Layered recursion, where a single object definition is called several times from within another definition, which in turn gets called several times from within yet another definition, allows construction of incredibly detailed objects with only a very small amount of code.

Technique

Eric Deren created a recursive object in POV-Ray that kicked off all sorts of other iterative animations. The trick here is to define an object, call it several times from within another object, call the resulting object several times from within a third object, and so on. Pretty soon, no more free memory. Then you reboot.

Steps

The following program (RECBOX.BAS) generates Eric's iterative box without using a ray tracer. It uses some hard-coded box drawing code that runs very fast:

```
RECBOX - Recursive Box -
  Reads a unit cell, and does a hard-coded turtle graphics
  double paint deal to show it in fake 3D. Habit forming.
  Use only as directed.
DECLARE SUB boxer (m, n, o, k)
SCREEN 12
DIM red(16), green(16), blue(16)
FOR y = 1 TO 4
      FOR x = 1 TO 4
              colornum = x + ((y - 1) * 4) - 1
              READ red(colornum), green(colornum), blue(colornum)
              KOLOR = 65536 * blue(colornum) + 256 * green(colornum) +
red(colornum)
              PALETTE colornum, KOLOR
              COLOR colornum
      NEXT x
NEXT y
'rainbow palette
DATA 0, 0, 0
DATA 16, 16, 16
DATA 32, 32, 32
DATA 0, 0, 0
DATA 63, 32, 0
DATA 58, 56, 0
DATA 16, 42, 0
DATA 0, 30, 36
```

```
DATA 0, 20, 40
DATA 0, 10, 48
DATA 0, 0, 63
DATA 20, 0, 53
DATA 23, 0, 29
DATA 19, 7, 17
DATA 50, 40, 45
DATA 63, 63, 63
WINDOW (-32, -16)-(32, 32) 'a,b,c
'WINDOW (-96, -48)-(96, 96) ' a,b,c,d
units = 3
pi = 3.14159
n = 1
m = 20
FOR z = 1 TO units
  FOR y = 1 TO units
         FOR x = 1 TO units
                READ state(x, y, z)
                n = n + 1
         NEXT x
  NEXT y
NEXT z
LOCATE 4, 68: PRINT "Unit Cell"
' Pick your unit cell here
' the long way - 0 = empty, the color of anything else is set by a cube
' number - anything over 8 or 12 is white
' iteration makes 'em add
DATA 1,1,1
DATA 1,0,1
DATA 1,1,1
DATA 1,0,1
DATA 0,0,0
DATA 1,0,1
DATA 1,1,1
DATA 1,0,1
DATA 1,1,1
' show the patterns
y1 = -5
y2 = 5
y3 = 15
x1 = 20
scale = 2
FOR x = 1 TO units
 FOR y = 1 TO units
```

CHAPTER EIGHT

```
continued from previous page
         FOR z = 1 TO units
        ax = x * scale
        ay = y * scale
         az = z * scale
        LINE (ax + x1, ay + y1)-(ax + x1 + scale, ay + y1 + scale), state(x, \Leftarrow
y, 1) + 3, BF
        LINE (ax + x1, ay + y2)-(ax + x1 + scale, ay + y2 + scale), state(x, \Leftarrow
y, 2) + 3, BF
        LINE (ax + x1, ay + y3)-(ax + x1 + scale, ay + y3 + scale), state(x, \Leftarrow
y, 3) + 3, BF
        NEXT z
  NEXT y
NEXT x
FOR ax = 1 TO units
FOR ay = 1 TO units
  FOR az = 1 TO units
        IF state(ax, ay, az) <> 0 THEN
         FOR bx = 1 TO units
           FOR by = 1 TO units
             FOR bz = 1 TO units
                IF state(bx, by, bz) <> 0 THEN
                 FOR cx = 1 TO units
                   FOR cy = 1 TO units
                    FOR cz = 1 TO units
                     IF state(cx, cy, cz) <> 0 THEN
                       FOR dx = 1 TO units
                         FOR dy = 1 TO units
                            FOR dz = 1 TO units
                             IF state(dx, dy, dz) <> 0 THEN
                                        x = ax + bx / units + cx / (units ^ 2) \leftarrow
+ dx / (units ^ 3)
                                        y = ay + by / units + cy / (units ^ 2) \Leftarrow
+ dy / (units ^ 3)
                                        z = az + bz / units + cz / (units ^ 2) \Leftarrow
+ dz / (units ^ 3)
                                        k = state(ax, ay, az) + state(bx, by, ⇐
bz) + state(cx, cy, cz)
                                        CALL boxer(x * 9, y * 9, z * 9, k)
                                         'DO WHILE INKEY$ = "": LOOP
                                        'LINE (x, y)-(x + 1 / units ^ 3, y + 1 / \Leftarrow
units ^ 3), state(ax, ay, az) + state(bx, by, bz) + state(cx, cy, cz) + ←
state(dx, dy, dz), BF
```

```
END IF
                         NEXT dz
                        NEXT dy
                        NEXT dx
                        END IF
                     NEXT cz
                   NEXT cy
                 NEXT cx
               END IF
            NEXT bz
          NEXT by
        NEXT bx
      END IF
  NEXT az
 NEXT ay
NEXT ax
SUB boxer (m, n, o, k)
       ' m,n,o are integer offsets, units in x, y, and z
       ' k is a color, can be useful, can be distracting
        ' sorry about all the numbers... hard coding them increased the
       ' draw speed by a factor of 3
       ' translate 3D m, nn and o into x & y screen offsets
       x = m * .545667 - o * .836517
       y = (n - m) * .901221 + m * .545667 - o * .224144
  ' frame the whole the cube will go in
  LINE (.547667 + x, -.370891 + y) - (1.384184 + x, -.146747 + y), 1
  LINE (1.384184 + x, -.146747 + y) - (1.401521 + x, .754474 + y), 1
  LINE (.853854 + x, 1.125365 + y)-(1.401521 + x, .754474 + y), 1
  LINE (.853854 + x, 1.125365 + y) - (.017337 + x, .901221 + y), 1
  LINE (.017337 + x, .901221 + y) - (x, y), 1
  LINE (x, y)-(.547667 + x, -.370891 + y), 1
  ' fill it with one color - erases the background
  PAINT (.7007 + x, .3772 + y), 2, 1
  ' draw and fill the cube
                                                                   continued on next page
```

continued from previous page

```
LINE (.547667 + x, -.370891 + y)-(1.384184 + x, -.146747 + y), 3
LINE (1.384184 + x, -.146747 + y)-(1.401521 + x, .754474 + y), 3
LINE (1.401521 + x, .754474 + y)-(.565005 + x, .53033 + y), 3
LINE (.565005 + x, .53033 + y)-(.547667 + x, -.370891 + y), 3
PAINT (.974594 + x, .191791 + y), 2 + k, 3
LINE (.853854 + x, 1.125365 + y)-(1.401521 + x, .754474 + y), 3
LINE (.017337 + x, .901221 + y)-(.565005 + x, .53033 + y), 3
LINE (.853854 + x, 1.125365 + y)-(.017337 + x, .901221 + y), 3
PAINT (.618151 + x, .889663 + y), 4 + k, 3
LINE (.017337 + x, .901221 + y)-(x, y), 3
LINE (x, y)-(.547667 + x, -.370891 + y), 3
PAINT (.141251 + x, .265165 + y), 6 + k, 3
```

END SUB

When you run the code, you end up with a box like the one shown in Figure 8-3.

We've listed this simulation code because it illustrates how nested loops can generate more and more detail. The last layer has been commented out, but provided you had enough screen resolution and memory, you could carry the recursive process on indefinitely.

Now let's switch to the Polyray file ITERATE.PI. This object is an interesting three-level recursive cube. The camera bobs up and down as it rotates on a spinning platform, giving a good sense of the spacial orientation in the interior of the box:

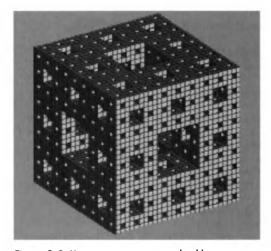


Figure 8-3 Menger sponge iterative cubical box

```
// ITERATE.PI
// Iterative Box Generator
start frame O
end frame 89
total_frames 90
outfile "iter"
define pi 3.1415927
define rad pi/180
define vy 22.5 + 22.5*cos((frame+12)*4*rad)
// set up the camera
viewpoint {
   from <75, vy, -65>
   at <0,-5,0>
   up <0,1,0>
   angle 45
   hither 1
   aspect 1.433
   resolution 320,200
   }
// get various surface finishes
background MidnightBlue
light <1, 1, 1>, < 60, 50, -50>
light <0.75, 0.75, 0.75>, < 0, 50, -15>
light <1, 1, 1>, < 0, 0, 0>
define Purple <0.714,0.427,0.639>
define BluGre <0.075,0.760,0.64 >
define BozoFunk
texture {
   noise surface {
      color white
      position_fn 1
      lookup_fn 1
      octaves 7
      turbulence 3
      ambient 0.2
      diffuse 0.6
      specular 0.3
      microfacet Reitz 5
      color map(
        [0.00, 0.35, BluGre, 0.000, BluGre, 0.000 ]
        [0.35, 0.40, BluGre, 0.000, Purple, 0.000]
        [0.40, 0.65, Purple, 0.000, Purple, 0.000]
        [0.65, 0.90, Purple, 0.000, BluGre, 0.000]
```

```
continued from previous page
        [0.90, 1.01, BluGre, 0.000, BluGre, 0.000 ])
    scale <25,25,25>
}
//---- Moss Greens & Tan
define Grnt25
texture {
   noise surface {
      color white
      position fn 1
      lookup_fn 1
      octaves 6
      turbulence 5
      ambient 0.2
      diffuse 0.6
      specular 0.3
      microfacet Reitz 5
      color_map(
        [0.000, 0.168, <0.824, 0.725, 0.584>, 0.000,
                      <0.514, 0.584, 0.533>, 0.000]
        [0.168, 0.301, <0.514, 0.584, 0.533>, 0.000,
                      <0.298, 0.376, 0.318>, 0.000]
        [0.301, 0.398, <0.298, 0.376, 0.318>, 0.000,
                      <0.263, 0.337, 0.282>, 0.000]
        [0.398, 0.558, <0.263, 0.337, 0.282>, 0.000,
                      <0.431, 0.506, 0.451>, 0.000]
        [0.558, 0.655, <0.431, 0.506, 0.451>, 0.000,
                      <0.529, 0.631, 0.471>, 0.000]
        [0.655, 0.735, <0.529, 0.631, 0.471>, 0.000,
                      <0.333, 0.376, 0.318>, 0.000]
        [0.735, 0.823, <0.333, 0.376, 0.318>, 0.000,
                      <0.298, 0.376, 0.318>, 0.000]
        [0.823, 0.876, <0.298, 0.376, 0.318>, 0.000,
                      <0.416, 0.376, 0.318>, 0.000]
        [0.876, 0.929, <0.416, 0.376, 0.318>, 0.000,
                      <0.416, 0.376, 0.318>, 0.000]
        [0.929, 1.001, <0.416, 0.376, 0.318>, 0.000,
                      <0.824, 0.725, 0.584>, 0.0001)
   }
   scale <25,25,25>
}
// set up background color & lights
include "\PLY\COLORS.INC"
// define a box made out of progressively smaller boxes
define cheese
object {
     object { box <-3, -3, -3>, <3, 3> }
   - object { box <-3.01, -1, -1>, <3.01, 1, 1>}
   - object { box <-1, -3.01, -1>, <1, 3.01, 1>}
   - object { box <-1, -1, -3.01>, <1, 1, 3.01>}
```

} define cheese2 object { object { cheese translate <-6,-6, -6> } + object { cheese translate < 0,-6, -6> } + object { cheese translate < 6,-6, -6> } + object { cheese translate <-6, 0, -6> } // object { cheese translate < 0, 0, -6 > } + object { cheese translate < 6, 0, -6> } + object { cheese translate <-6, 6, -6> } + object { cheese translate < 0, 6, -6> } + object { cheese translate < 6, 6, -6> } + object { cheese translate <-6,-6, 0> } // object { cheese translate < 0,-6, 0> } + object { cheese translate < 6,-6, 0> } // object { cheese translate <-6, 0, 0> } // object { cheese translate < 0, 0, 0> } // object { cheese translate < 6, 0, 0> } + object { cheese translate <-6, 6, 0> } // object { cheese translate < 0, 6, 0> } + object { cheese translate < 6, 6, 0> } + object { cheese translate <-6,-6, 6> } + object { cheese translate < 0,-6, 6> } + object { cheese translate < 6,-6, 6> } + object { cheese translate <-6, 0, 6> } // object { cheese translate < 0, 0, 6> } + object { cheese translate < 6, 0, 6> } + object { cheese translate <-6, 6, 6> } + object { cheese translate < 0, 6, 6> } + object { cheese translate < 6, 6, 6> } define cheese3 object { object { cheese2 translate <-18,-18, -18> } + object { cheese2 translate < 0,-18, -18> } + object { cheese2 translate < 18,-18, -18> } + object { cheese2 translate <-18, 0, -18> } // object { cheese2 translate < 0, 0, -18> } + object { cheese2 translate < 18, 0, -18> } + object { cheese2 translate <-18, 18, -18> }

```
continued from previous page
+ object { cheese2 translate < 0, 18, -18> }
+ object { cheese2 translate < 18, 18, -18> }
+ object { cheese2 translate <-18,-18, 0> }
// object { cheese2 translate < 0,-18, 0> }
+ object { cheese2 translate < 18,-18, 0> }
// object { cheese2 translate <-18, 0, 0> }
// object { cheese2 translate < 0, 0, 0> }
// object { cheese2 translate < 18, 0, 0> }
+ object { cheese2 translate <-18, 18, 0> }
// object { cheese2 translate < 0, 18, 0> }
+ object { cheese2 translate < 18, 18, 0> }
+ object { cheese2 translate <-18,-18, 18> }
+ object { cheese2 translate < 0,-18, 18> }
+ object { cheese2 translate < 18,-18, 18> }
+ object { cheese2 translate <-18, 0, 18> }
// object { cheese2 translate < 0, 0, 18> }
+ object { cheese2 translate < 18, 0, 18> }
+ object { cheese2 translate <-18, 18, 18> }
+ object { cheese2 translate < 0, 18, 18> }
+ object { cheese2 translate < 18, 18, 18> }
object {cheese3 Grnt25 rotate <0, frame*4,0>}
// create a ground plane
object {
   disc <0, -27.001, 0>, <0, 1, 0>, 0, 100
   BozoFunk
   rotate <0, frame *4,0>
```

A sample image is shown in Figure 8-4.

How It Works

The initial object is called *cheese*, I suppose because this is a little like Swiss cheese. It is a box, with three rectangular bites out of it, each 90° to one another:

```
define cheese
object {
    object { box <-3, -3, -3>, <3, 3, 3> }
    - object { box <-3.01, -1, -1>, <3.01, 1, 1>}
    - object { box <-1, -3.01, -1>, <1, 3.01, 1>}
    - object { box <-1, -1, -3.01>, <1, 1, 3.01>}
}
```

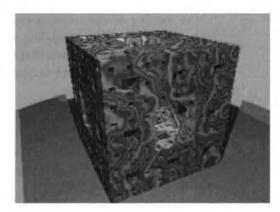


Figure 8-4 Iterative cube on a platter

It forms the basis of the next layer, which calls 27 copies of it to form a second cube three cheese units on a side. As in the unit cell display in RECBOX.BAS, we manually turn off certain cubes by commenting out their definitions:

```
define cheese2
object {
   object { cheese translate <-6,-6, -6> }
   + object { cheese translate < 0,-6, -6> }
   + object { cheese translate < 6,-6, -6> }

   + object { cheese translate < -6, 0, -6> }

// object { cheese translate < 0, 0, -6> } <- this one's gone
   + object { cheese translate < 6, 0, -6> }
```

Cheese2 forms the basis for the next layer, and really all we have to do is a block copy of this layer and change the names from cheese to cheese2. We would have continued this process even further, but we'd run out of memory, even on an 8MB machine.



8.3 How do I...

Maximize the use of system memory when building recursive objects?

You'll find the code for this in: PLY\CHAPTER8\BORG

Problem

The iterative box in Section 8.2 can only be pushed to three levels before the memory requirements simply become too great for the ray tracer (or more correctly, the system's memory) to cope with. Even on a system with 32MB,

the number of individual items simply grows too large. But looking at the box, we're really only dealing with a tiny part that's visible—the outside. If we can cut away the parts that don't show, maybe we can get the item count down to where it doesn't exceed the system's capacity.

Warning

A word of caution: if you get the message "Failed to allocate CSG node" or "Cannot allocate transformation," you're going to have to either get more memory or make a simpler model. This animation will not run in 16MB RAM—you need 32MB.

Technique

An interesting result of cutting away details is that it's an iterative process too. Four levels are shown in the following Polyray code, and // comment lines delete the parts we don't need. We use the tumbling code of Section 2.6, here in its simplest form, to orbit the iterative box and show the details of its construction.

Steps

There are two steps that extend the previous section's code to this one.

- 1. Copy the third layer, paste it below the previous code, and edit references to *cheese2* to read *cheese3*. This builds the next layer.
- 2. Go through the code and remove all those items that are not visible from the camera's vantage point.

This step requires looking at a sketch of the cube and, taking each of the 27 members of a layer, determining which ones won't be visible from that perspective. All we need to do is keep the elements for the three sides facing the camera, which without duplication amounts to the nine top members, the six lower members on one side, and four members on the other, for a total of 19. Since each side is already missing its middle member, that drops the count to 16. Here is the code for this (ITER*.PI).

```
//ITER*.PI
// Iterative Box Deal
start_frame 0
end_frame 359
total_frames 360
define index 360/total_frames
```

```
outfile "iter"
define pi 3.14159
define rad pi/180
define ang1 (frame*index + 0) * rad
define ang2 (frame*index + 60) * rad
define ang3 (frame*index + 120) * rad
define vx 300*SIN(ang1)
define vy 300*SIN(ang2)
define vz 300*SIN(ang3)
// set up the camera
viewpoint {
   from <vx,vy,vz>
   at <0,-30,0>
   up <0,1,0>
   angle 40
   hither 1
   aspect 1.433
   resolution 320,200
// get various surface finishes
background MidnightBlue
light <1, 1, 1>, < 180, 150, -150>
light <0.75, 0.75, 0.75>, < 0, 100, -15>
light <1, 1, 1>, < 0, 0, 0>
// set up background color & lights
include "\PLY\COLORS.INC"
// define a short pyramid made out of progressively smaller boxes
define cheese
object {
     object { box <-3, -3, -3>, <3, 3> }
   - object { box <-3.01, -1, -1>, <3.01, 1, 1>}
   - object { box <-1, -3.01, -1>, <1, 3.01, 1>}
   - object { box <-1, -1, -3.01>, <1, 1, 3.01>}
define cheese2
object {
     object { cheese translate <-6,-6, -6> }
   + object { cheese translate < 0,-6, -6> }
   + object { cheese translate < 6,-6, -6> }
   + object { cheese translate <-6, 0, -6> }
//// object { cheese translate < 0, 0, -6 > }
   + object { cheese translate < 6, 0, -6> }
```

```
continued from previous page
   + object { cheese translate <-6, 6, -6> }
   + object { cheese translate < 0, 6, -6> }
   + object { cheese translate < 6, 6, -6> }
// + object { cheese translate <-6,-6, 0> }
//// object { cheese translate < 0,-6, 0> }
   + object { cheese translate < 6,-6, 0> }
//
//// object { cheese translate <-6, 0, 0> }
//// object { cheese translate < 0, 0, 0> }
//// object { cheese translate < 6, 0, 0> }
   + object { cheese translate <-6, 6, 0> }
//// object { cheese translate < 0, 6, 0> }
   + object { cheese translate < 6, 6, 0> }
//
//
// + object { cheese translate <-6,-6, 6> }
// + object { cheese translate < 0,-6, 6> }
   + object { cheese translate < 6,-6, 6> }
// + object { cheese translate <-6, 0, 6> }
//// object { cheese translate < 0, 0, 6> }
   + object { cheese translate < 6, 0, 6> }
   + object { cheese translate <-6, 6, 6> }
   + object { cheese translate < 0, 6, 6> }
   + object { cheese translate < 6, 6, 6> }
}
define cheese3
object {
     object { cheese2 translate <-18,-18, -18> }
   + object { cheese2 translate < 0,-18, -18> }
   + object { cheese2 translate < 18,-18, -18> }
   + object { cheese2 translate <-18, 0, -18> }
//// object { cheese2 translate < 0, 0, -18 > }
   + object { cheese2 translate < 18, 0, -18> }
   + object { cheese2 translate <-18, 18, -18> }
   + object { cheese2 translate < 0, 18, -18> }
   + object { cheese2 translate < 18, 18, -18> }
// + object { cheese2 translate <-18,-18, 0> }
//// object { cheese2 translate < 0,-18, 0> }
   + object { cheese2 translate < 18,-18, 0> }
11
//// object { cheese2 translate <-18, 0, 0> }
//// object { cheese2 translate < 0, 0, 0> }
```

```
//// object { cheese2 translate < 18, 0, 0> }
11
   + object { cheese2 translate <-18, 18, 0> }
//// object { cheese2 translate < 0, 18, 0> }
  + object { cheese2 translate < 18, 18, 0> }
11
11
// + object { cheese2 translate <-18,-18, 18> }
// + object { cheese2 translate < 0,-18, 18> }
   + object { cheese2 translate < 18,-18, 18> }
// + object { cheese2 translate <-18, 0, 18> }
//// object { cheese2 translate < 0, 0, 18> }
   + object { cheese2 translate < 18, 0, 18> }
   + object { cheese2 translate <-18, 18, 18> }
   + object { cheese2 translate < 0, 18, 18> }
   + object { cheese2 translate < 18, 18, 18> }
}
define cheese4
object {
    object { cheese3 translate <-54,-54, -54> }
   + object { cheese3 translate < 0,-54, -54 >  }
   + object { cheese3 translate < 54,-54, -54> }
   + object { cheese3 translate <-54, 0, -54> }
// object { cheese3 translate < 0, 0, -54 >  }
  + object { cheese3 translate < 54, 0, -54> }
   + object { cheese3 translate <-54, 54, -54> }
   + object { cheese3 translate < 0, 54, -54> }
   + object { cheese3 translate < 54, 54, -54> }
// + object { cheese3 translate <-54,-54,</pre>
                                            0> }
                                            0> }
//// object { cheese3 translate < 0,-54,
   + object { cheese3 translate < 54,-54,
                                            0> }
//// object { cheese3 translate <-54, 0,
                                            0> }
//// object { cheese3 translate < 0, 0,
                                            0> }
//// object { cheese3 translate < 54, 0,
                                            0> }
11
   + object { cheese3 translate <-54, 54,
                                            0> }
//// object { cheese3 translate < 0, 54,
                                           0> }
                                           0> }
   + object { cheese3 translate < 54, 54,
11
11
    object { cheese3 translate <-54,-54, 54> }
//
// + object { cheese3 translate < 0,-54, 54> }
   + object { cheese3 translate < 54,-54, 54> }
```

```
continued from previous page
// + object { cheese3 translate <-54, 0,</pre>
    object { cheese3 translate < 0, 0,
                                            54> }
   + object { cheese3 translate < 54, 0,
                                            54> }
   + object { cheese3 translate <-54, 54,
                                            54> }
   + object { cheese3 translate < 0, 54,
   + object { cheese3 translate < 54, 54, 54> }
}
object {cheese4 shiny_coral rotate <180,-90,0> }
include "map"
// Create a ground plane
object {
   sphere <0,0,0>, 10*54
   scream_texture
```

How It Works

We've already covered the iterative aspects. The only unique part here is that we figure out which parts aren't really necessary for generating the outside surface of the box, and comment those lines out. The camera motion orbits the cube to let us see all six sides, and the back side that's pretty much eaten away is actually far more interesting than the front sides we originally set out to make (see Figure 8-5).

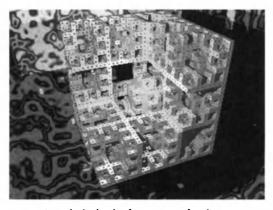


Figure 8-5 The back side of our iterative facade



🔑 8.4 How do I...

Make a recursive, three-level morphing pat of butter?

You'll find the code for this in: PLY\CHAPTER8\BUTTER

Problem

Really strange objects that do severely nonlinear stuff are often based on layering simple motions until they appear more complicated than they are. Such is the case with the following animation. It started out as a simple three-level propeller and ended up with lozenges coming out of it in all directions.

Technique

This animation makes use of recursion to transform propeller metaballs into a really odd sloshing pat of butter with delusions of grandeur. The odd thing is that it's actually fairly simple to model. Take a cross. At the midpoints of each branch, place another cross, a smaller one that will fit in the space provided. Now do it one more time, for a total of 4+16+64... 84 crosses. Place metaballs at the ends of each branch and start rotating the crosses about their centers. Now displace each level from the next one, in three phases. This is shown in Figure 8-6.

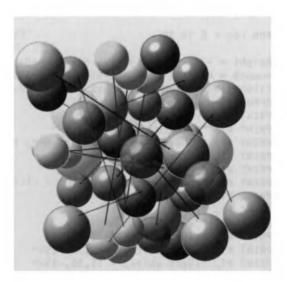


Figure 8-6 Recursive rotating blob object

Steps

A direct translation of this code to Polyray would have been difficult, given all the loops and subroutines, so the animation is all done with external files. However, a rewrite based on the principles covered in the next section would be fairly simple to do.

Run the following QuickBasic code (BUTTER.BAS), which generates a series of 90 Polyray data files containing the scene setups and the definitions for our evolving blob:

```
'BUTTER.BAS
DECLARE SUB rotate (a, b, c)
DECLARE SUB branch (a, b, c, d)
TYPE vector
  a AS SINGLE
  b AS SINGLE
  c AS SINGLE
  d AS SINGLE
END TYPE
DIM br(1024) AS vector
COMMON SHARED br() AS vector, angle, rad, n, depth, rot, dir
SCREEN 12
WINDOW (-1.6, -1.2)-(1.6, 1.2)
pi = 3.1415927#
rad = pi / 180
dir = 1
FOR rot = 0 TO 89
height = -.35 + .1 * SIN((4 * rot + 60) * rad)
count$ = RIGHT$("000" + LTRIM$(STR$(rot)), 3)
filename$ = "blobr" + count$ + ".pi"
OPEN filename$ FOR OUTPUT AS #1
PRINT #1, "viewpoint {"
PRINT #1, " from <1.5,2.0,-2.5>"
PRINT #1, USING " at <0,##.####,0>"; height
PRINT #1, " up <0,1,0>"
PRINT #1, " angle 30"
PRINT #1, " resolution 320,200"
PRINT #1, "
             aspect 1.433"
PRINT #1, "
PRINT #1, ""
PRINT #1, "background MidnightBlue"
PRINT #1, "light white, <-15,30,-25>"
PRINT #1, "light white, < 15,30,-25>"
PRINT #1, ""
PRINT #1, "object {"
```

```
PRINT #1, " blob 0.10:"
angle = 90
n = 1
br(n).a = -1
br(n).b = 0
br(n).c = 1
br(n).d = 0
s = 1
dir = 1
CLS
FOR depth = 1 \text{ TO } 3
        'dir = 1
        LOCATE 1, 1: PRINT "frame="; rot
        olds = n
        FOR z = s TO n
               CALL branch(br(z).a, br(z).b, br(z).c, br(z).d)
               dir = -dir
        NEXT z
        s = olds + 1
NEXT depth
PRINT #1, USING " ##.####, ##.###, <##.####, ##.####, ##.###.##."; r, .225, ←
br(n).c, -depth / 10, br(n).d
PRINT #1, "
             u_steps 20"
PRINT #1, "
             v_steps 20"
PRINT #1, "
             texture {"
PRINT #1, "
               surface {"
PRINT #1, "
                   ambient Gold, 0.2"
PRINT #1, "
                   diffuse Gold, 0.8"
PRINT #1, "
                   specular white, 0.6"
                 microfacet Phong 5"
reflection white, 0.25"
transmission white, 0, 0"
PRINT #1, "
PRINT #1, "
PRINT #1, "
PRINT #1, "
                    }"
PRINT #1, "
                }"
PRINT #1, USING " rotate <0.0, ###.#,0.0>"; -rot
PRINT #1, " }"
CLOSE #1
NEXT rot
OPEN "doit.bat" FOR OUTPUT AS #2
FOR rot = 0 TO 89
  count$ = RIGHT$("000" + LTRIM$(STR$(rot)), 3)
  PRINT #2, "call pr blobr" + count$
NEXT rot
CLOSE #2
SUB branch (a, b, c, d)
        centerx = (a + c) / 2
       centery = (b + d) / 2
```

```
continued from previous page
       length = .95 ^ depth * ((a - c) ^ 2 + (b - d) ^ 2) ^ .5
       x = (a - c) + .000001
       y = b - d
       t = ATN(y / x) / rad
       IF x < 0 THEN t = t + 180
       IF t < 0 THEN t = t + 360
       ph = 45
       height = -.2 + .15 * SIN((4 * rot + 120 * depth) * rad)
FOR ang = -t + depth * rot * dir TO 360 - angle - t + depth * rot * dir + 1 \Leftarrow
STEP angle
       n = n + 1
       br(n).a = centerx
       br(n).b = centery
       br(n).c = length * SIN(ang * rad) / 2 + centerx
       br(n).d = length * COS(ang * rad) / 2 + centery
  x1 = br(n).a
  y1 = br(n).b
  z1 = height
  CALL rotate(x1, y1, z1)
 x2 = br(n).c
 y2 = br(n).d
  z2 = height
  CALL rotate(x2, y2, z2)
       LINE (x1, z1)-(x2, z2), depth
               r = length * depth / 8
  x = br(n).c
  y = br(n).d
  z = height
  CALL rotate(x, y, z)
       CIRCLE (x, z), r
       PRINT #1, USING " ##.###, ##.##, <##.###, ##.###,##.###,#.###>,"; r, ←
.225, br(n).c, height, br(n).d
       'LINE -(br(n).c, br(n).d), depth
NEXT ang
END SUB
SUB rotate (x, y, z)
  xrotate = 90
 yrotate = 0
  zrotate = 0
```

```
z0 = z
       x1 = x0
       y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
       z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)
       x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
       z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)
       x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)
       y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)
       z3 = z2
       x = x3
       y = y3
       z = z3
END SUB
             This code generates a series of data files that look like this:
viewpoint {
   from <1.5,2.0,-2.5>
   at <0,-0.2601,0>
   up <0,1,0>
   angle 30
   resolution 320,200
   aspect 1.433
   }
background MidnightBlue
light white, <-15,30,-25>
light white, < 15,30,-25>
object {
   blob 0.10:
   0.2375, 0.225, <-0.0166, -0.0756, -0.9499>,
   0.2375, 0.225, <-0.9499, -0.0756, 0.0166>,
   0.2375, 0.225, < 0.0166, -0.0756, 0.9499>,
   0.2375, 0.225, < 0.9499, -0.0756, -0.0166>,
   0.2143, 0.225, <-0.4369, -0.3348, -0.4824>,
   0.2143, 0.225, <-0.0158, -0.3348, -0.0463>,
   0.2143, 0.225, < 0.4203, -0.3348, -0.4674>,
   0.2143, 0.225, <-0.0008, -0.3348, -0.9035>,
    . . .
   0.1378, 0.225, < 0.8717, -0.1895, -0.0387>,
   0.0000, 0.225, < 0.8717, -0.4000, -0.0387>
   u_steps 20
   v_steps 20
```

x0 = xy0 = y

```
texture {
    surface {
        ambient Gold, 0.2
        diffuse Gold, 0.8
        specular white, 0.6
        microfacet Phong 5
        reflection white, 0.25
        transmission white, 0, 0
      }
}
rotate <0.0, -1.0,0.0>
}
```

How It Works

Each level of metaballs in this blob has slightly different strength values, making each level a bit easier to spot as the animation develops. During the course of the animation, various groups of metaballs gang up and protrude from the surface, then dive back in and get lost in the shuffle. The rotational code is included in the BUTTER.BAS as a parameter that is handled every frame. A sample frame from the animation is shown in Figure 8-7.

Differentiating the Image

In BUTTER2.BAS, the entire blob is a single color. It would be more interesting if each level had its own color. The following code produces the data file for this:

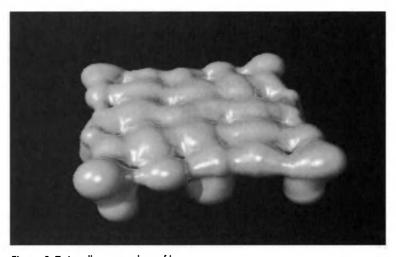


Figure 8-7 A really possessed pat of butter

```
'BUTTER2.BAS
DECLARE SUB rotate (a, b, c)
DECLARE SUB branch (a, b, c, d)
TYPE vector
  a AS SINGLE
  b AS SINGLE
  c AS SINGLE
  d AS SINGLE
END TYPE
DIM br(1024) AS vector
COMMON SHARED br() AS vector, angle, rad, n, depth, rot, dir
SCREEN 12
WINDOW (-1.6, -1.2)-(1.6, 1.2)
pi = 3.1415927#
rad = pi / 180
dir = 1
FOR rot = 0 TO 89
       height = -.35 + .1 * SIN((4 * rot + 60) * rad)
  count$ = RIGHT$("000" + LTRIM$(STR$(rot)), 3)
  filename$ = "blobr" + count$ + ".pi"
  OPEN filename$ FOR OUTPUT AS #1
PRINT #1, "viewpoint {"
PRINT #1, " from <1.5,2.0,-2.5>"
PRINT #1, USING " at <0,##.####,0>"; height
PRINT #1, "
            up <0,1,0>"
PRINT #1, "
            angle 30"
PRINT #1, " resolution 320,200"
PRINT #1, "
            aspect 1.433"
PRINT #1, "
PRINT #1, ""
PRINT #1, "background MidnightBlue"
PRINT #1, "light white, <-15,30,-25>"
PRINT #1, "light white, < 15,30,-25>"
PRINT #1, ""
PRINT #1, "object {"
PRINT #1, " blob 0.10:"
angle = 90
n = 1
br(n).a = -1
br(n).b = 0
br(n).c = 1
br(n).d = 0
s = 1
dir = 1
CLS
```

```
continued from previous page
FOR depth = 1 \text{ TO } 3
        'dir = 1
        LOCATE 1, 1: PRINT "frame="; rot
        olds = n
        FOR z = s TO n
               CALL branch(br(z).a, br(z).b, br(z).c, br(z).d)
                dir = -dir
        NEXT z
        s = olds + 1
  'DO WHILE INKEY$ = "": LOOP
NEXT depth
PRINT #1, USING " ##.###, ##.###, <##.###,##.####,##.###>"; r, .225, ←
br(n).c, -depth / 10, br(n).d
PRINT #1, " u_steps 20"
PRINT #1, "
              v_steps 20"
PRINT #1, " texture {"
PRINT #1, "
PRINT #1, "
               surface {"
                ambient Coral, 0.2"
diffuse Coral, 0.8"
specular white, 0.6"
microfacet Phong 5"
reflection white, 0.25"
transmission white, 0, 0"
PRINT #1, "
                    }"
PRINT #1, "
PRINT #1, USING " rotate <0.0, ####.#,0.0>"; -rot + 45 * SIN(rot * 4 * rad)
PRINT #1, " }"
PRINT #1,
n = 1
s = 1
dir = 1
depth = 1
PRINT #1, "object {"
PRINT #1, " blob 0.09:"
        LOCATE 1, 1: PRINT "frame="; rot
        olds = n
        FOR z = s TO n
                CALL branch(br(z).a, br(z).b, br(z).c, br(z).d)
                dir = -dir
        NEXT z
        s = olds + 1
PRINT #1, USING " ##.###, ##.###, <##.###,##.####,"; r, .225, ←
br(n).c, -depth / 10, br(n).d
PRINT #1, " u_steps 20"
PRINT #1, "
              v_steps 20"
PRINT #1, " texture {"
PRINT #1, "
               surface {"
PRINT #1, "
                    ambient Maroon, 0.2"
```

```
PRINT #1, "
                   diffuse Maroon, 0.8"
PRINT #1, "
                   specular white, 0.6"
PRINT #1, "
                   microfacet Phong 5"
PRINT #1, "
                   reflection white, 0.25"
PRINT #1, "
                    transmission white, 0, 0"
PRINT #1, "
PRINT #1, "
PRINT #1, USING " rotate <0.0, ####.#,0.0>"; -rot + 45 * SIN(rot * 4 * rad)
PRINT #1, "
             }"
PRINT #1,
depth = 2
PRINT #1, "object {"
PRINT #1, " blob 0.09:"
       LOCATE 1, 1: PRINT "frame="; rot
       olds = n
       FOR z = s TO n
               CALL branch(br(z).a, br(z).b, br(z).c, br(z).d)
               dir = -dir
       NEXT z
       s = olds + 1
PRINT #1, USING " ##.###, ##.###, <##.####,##.###,##.###>"; r, .225, ←
br(n).c, -depth / 10, br(n).d
PRINT #1, " u_steps 20"
PRINT #1, "
              v_steps 20"
PRINT #1, " texture {"
             surface {"
PRINT #1, "
                ambient DarkSlateBlue, 0.2"
PRINT #1, "
PRINT #1, "
                  diffuse DarkSlateBlue, 0.8"
                 specular white, 0.6"
microfacet Phong 5"
reflection white, 0.25"
transmission white, 0, (
PRINT #1, "
PRINT #1, "
PRINT #1, "
PRINT #1, "
                  transmission white, 0, 0"
PRINT #1, "
                   }"
PRINT #1, "
PRINT #1, USING " rotate <0.0, ####.#,0.0>"; -rot + 45 * SIN(rot * 4 * rad)
PRINT #1, " }"
PRINT #1,
depth = 3
PRINT #1, "object {"
PRINT #1, " blob 0.09:"
       LOCATE 1, 1: PRINT "frame="; rot
       olds = n
       FOR z = s TO n
               CALL branch(br(z).a, br(z).b, br(z).c, br(z).d)
               dir = -dir
       NEXT z
                                                                   continued on next page
```

```
continued from previous page
        s = olds + 1
PRINT #1, USING " ##.###, ##.###, <##.###, ##.###,##.###>"; r, .225, ←
br(n).c, -depth / 10, br(n).d
PRINT #1, "
              u_steps 20"
PRINT #1, "
              v_steps 20"
PRINT #1, " texture {"
PRINT #1, "
              surface {"
PRINT #1, "
                 ambient SteelBlue, 0.2"
                  diffuse SteelBlue, 0.8"
specular white, 0.6"
microfacet Phong 5"
reflection white, 0.25"
PRINT #1, "
                    transmission white, 0, 0"
PRINT #1, "
                     }"
PRINT #1, "
PRINT #1, USING "
                    rotate <0.0, ####.#,0.0>"; -rot + 45 * SIN(rot * 4 * rad)
PRINT #1, " }"
PRINT #1,
CLOSE #1
NEXT rot
OPEN "doit.bat" FOR OUTPUT AS #2
FOR rot = 0 TO 89
  count$ = RIGHT$("000" + LTRIM$(STR$(rot)), 3)
  PRINT #2, "call pr blobr" + count$
NEXT rot
CLOSE #2
SUB branch (a, b, c, d)
        centerx = (a + c) / 2
        centery = (b + d) / 2
        length = .95 ^ depth * ((a - c) ^ 2 + (b - d) ^ 2) ^ .5
        x = (a - c) + .000001
        y = b - d
        t = ATN(y / x) / rad
        IF x < 0 THEN t = t + 180
        IF t < 0 THEN t = t + 360
        ph = 45
        IF Dir > 0 THEN ph = angle / 2 ELSE ph = 0
        height = -.2 + .15 * SIN((4 * rot + 120 * depth) * rad)
FOR ang = -t + depth * rot * dir TO 360 - angle - t + depth * rot * dir + 1 \Leftarrow
STEP angle
        n = n + 1
        br(n).a = centerx
        br(n).b = centery
```

```
br(n).c = length * SIN(ang * rad) / 2 + centerx
       br(n).d = length * COS(ang * rad) / 2 + centery
  x1 = br(n).a
  y1 = br(n).b
  z1 = height
  CALL rotate(x1, y1, z1)
  x2 = br(n).c
  y2 = br(n).d
  z2 = height
  CALL rotate(x2, y2, z2)
       LINE (x1, z1)-(x2, z2), depth
       r = length * depth / 8
  x = br(n).c
  y = br(n).d
  z = height
  CALL rotate(x, y, z)
       CIRCLE (x, z), r
       PRINT #1, USING " ##.###, ##.###, <##.####,##.####,#.####>,"; r, ←
.225, br(n).c, height, br(n).d
NEXT and
END SUB
SUB rotate (x, y, z)
  xrotate = 90
  yrotate = 0
  zrotate = 0
       x0 = x
       y0 = y
       z0 = z
       x1 = x0
       y1 = y0 * COS(xrotate * rad) - z0 * SIN(xrotate * rad)
       z1 = y0 * SIN(xrotate * rad) + z0 * COS(xrotate * rad)
       x2 = z1 * SIN(yrotate * rad) + x1 * COS(yrotate * rad)
       y2 = y1
       z2 = z1 * COS(yrotate * rad) - x1 * SIN(yrotate * rad)
       x3 = x2 * COS(zrotate * rad) - y2 * SIN(zrotate * rad)
       y3 = x2 * SIN(zrotate * rad) + y2 * COS(zrotate * rad)
       z3 = z2
       x = x3
       y = y3
       z = z3
```

END SUB

439

This second program produces the following Polyray code:

```
viewpoint {
   from <1.5,2.0,-2.5>
   at <0,-0.2601,0>
   up <0,1,0>
   angle 30
   resolution 320,200
   aspect 1.433
background MidnightBlue
light white, <-15,30,-25>
light white, < 15,30,-25>
// main blob
object {
   blob 0.10:
    0.2375, 0.225, <-0.0166, -0.0756, -0.9499>,
    0.2375, 0.225, <-0.9499, -0.0756, 0.0166>,
    0.2375, 0.225, < 0.0166, -0.0756, 0.9499>,
    0.2375, 0.225, < 0.9499, -0.0756, -0.0166>,
    0.2143, 0.225, <-0.4369, -0.3348, -0.4824>,
    0.1378, 0.225, < 0.8717, -0.1895, -0.0387>,
    0.0000, 0.225, < 0.8717, -0.4000, -0.0387>
   u steps 20
   v_steps 20
   texture {
      surface {
         ambient Coral, 0.2
         diffuse Coral, 0.8
         specular white, 0.6
         microfacet Phong 5
         reflection white, 0.25
         transmission white, 0, 0
      }
   rotate <0.0,
                   2.1,0.0>
object {
   blob 0.09:
    0.2375, 0.225, <-0.0166, -0.0756, -0.9499>,
    0.2375, 0.225, <-0.9499, -0.0756, 0.0166>,
    0.2375, 0.225, < 0.0166, -0.0756, 0.9499>,
    0.2375, 0.225, < 0.9499, -0.0756, -0.0166>,
    0.0000, 0.225, < 0.9499, -0.1000, -0.0166>
   u_steps 20
   v_steps 20
   texture {
```

```
surface {
         ambient Maroon, 0.2
         diffuse Maroon, 0.8
         specular white, 0.6
        microfacet Phong 5
         reflection white, 0.25
         transmission white, 0, 0
         }
      }
   rotate <0.0,
                   2.1,0.0>
object {
  blob 0.09:
    0.2143, 0.225, <-0.4369, -0.3348, -0.4824>,
    0.2143, 0.225, <-0.0158, -0.3348, -0.0463>,
    0.2143, 0.225, < 0.4203, -0.3348, -0.4674>,
    0.2143, 0.225, <-0.0008, -0.3348, -0.9035>,
    0.2143, 0.225, <-0.4525, -0.3348, 0.4364>,
    0.2143, 0.225, <-0.0468, -0.3348, -0.0141>,
    0.2143, 0.225, <-0.4974, -0.3348, -0.4198>,
    0.2143, 0.225, <-0.9030, -0.3348, 0.0307>,
    0.2143, 0.225, < 0.4369, -0.3348, 0.4824>,
    0.2143, 0.225, < 0.0158, -0.3348, 0.0463>,
    0.2143, 0.225, <-0.4203, -0.3348, 0.4674>,
    0.2143, 0.225, < 0.0008, -0.3348, 0.9035>,
    0.2143, 0.225, < 0.4525, -0.3348, -0.4364>,
    0.2143, 0.225, < 0.0468, -0.3348, 0.0141>,
    0.2143, 0.225, < 0.4974, -0.3348, 0.4198>,
    0.2143, 0.225, < 0.9030, -0.3348, -0.0307>,
    0.0000, 0.225, < 0.9030, -0.2000, -0.0307>
   u_steps 20
   v_steps 20
   texture {
      surface {
         ambient DarkSlateBlue, 0.2
         diffuse DarkSlateBlue, 0.8
         specular white, 0.6
         microfacet Phong 5
         reflection white, 0.25
         transmission white, 0, 0
   rotate <0.0, 2.1,0.0>
object {
   blob 0.09:
    0.1378, 0.225, <-0.2354, -0.1895, -0.2953>,
    0.1378, 0.225, <-0.0393, -0.1895, -0.4658>,
    . . .
    0.1378, 0.225, < 0.6698, -0.1895, -0.2023>,
```

```
continued from previous page
    0.1378, 0.225, < 0.5062, -0.1895, -0.0003>,
    0.1378, 0.225, < 0.7082, -0.1895, 0.1633>,
    0.1378, 0.225, < 0.8717, -0.1895, -0.0387>,
    0.0000, 0.225, < 0.8717, -0.3000, -0.0387>
   u steps 20
   v_steps 20
   texture {
      surface {
         ambient SteelBlue, 0.2
         diffuse SteelBlue, 0.8
         specular white, 0.6
         microfacet Phong 5
         reflection white, 0.25
         transmission white, 0, 0
      }
   rotate <0.0,
                    2.1,0.0>
```

This code includes not only the object from our first animation, but each level separately declared as individual blobs, now colored Maroon, DarkSlateBlue and SteelBlue. The three levels of this blob are slightly different sizes, and they peek out from around one another at various times (Figure 8-8).

Batch Run

The only other thing that's needed is a batch file to call each data file sequentially. That is automatically generated by the QuickBasic code as a file called DOIT.BAT.

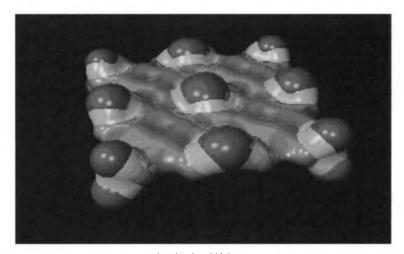


Figure 8-8 Four cavorting nested multicolored blobs



```
call pr blobr000
call pr blobr001
call pr blobr002
call pr blobr003
call pr blobr005
call pr blobr005
call pr blobr007
call pr blobr007
call pr blobr008
call pr blobr009
call pr blobr010
call pr blobr011
call pr blobr012
call pr blobr013
```



尾 8.5 How do I...

Make a pentagonal kaleidoscope?

You'll find the code for this in: PLY\CHAPTER8\KALE1

Problem

Kaleidoscopes are usually done with bits of colored glass and a triangular mirrored tunnel, which form a regularly repeating pattern that goes out in all directions. We can use recursion to make our kaleidoscope out of real objects, allowing us to create symmetries that aren't possible with mirrors. Mirrors will be used for creating extended reflections, not for generating symmetry.

Technique

We'll generate a header file for the static elements, then write a short QuickBasic program to define four successive layers of recursive elements. While you may not believe this, the entire recursive array (37 pages of code) was originally written using nothing more than macros in an editor and a series of block copies. You're being spared this task by our resorting to some simple loops in QuickBasic.

Steps

KALE1.PI, the Polyray header file for this, is standard:

```
//KALE1.PI - PENTAGONAL KALEIDOSCOPE
start_frame 0
end_frame 71
total_frames 72
outfile pr4
include "\PLY\COLORS.INC"
viewpoint {
   from <0.0.4>
   at <0,0,0>
   up <0,1,0>
   angle 35
   resolution 32,20
   aspect 1.433
   }
//background SkyBlue
light white, <0.45,0.45,4.5>
define pi 3.14159
define rad pi/180
define floor
object
  { polygon 4,<-5,-5,0>,<5,-5,0>,<-5,5,0> matte_blue }
define mir2
texture {
   surface {
      ambient white, 0.0
      diffuse white, 0.07
      specular O
      reflection white, 1
      }
   }
define reflective_gold texture { reflective { color gold } }
define reflective_goldenrod texture { reflective { color goldenrod } }
define reflective_navy texture { reflective { color navy } }
define reflective_lightsteelblue texture { reflective { color lightsteelblue }
}
define tip -1.5+sin(radians(frame*5))
light white * 0.75, < 0, 0, tip + 0.1>
object { cone <0,0,tip>,0,<0,0,5>,10 mir2 }
object { disc <0,0,5>,<0,0,-1>,10 mir2 }
define br frame // base rotation, was frame*5, now is just frame
include "kalei.inc"
```

The KALEI.INC file mentioned previously is generated with the following code:

```
OPEN "kalei.inc" FOR OUTPUT AS #1
PRINT #1, "define L0001 rotate(<1,0,0>,<0,0,br+360*1/5>)"
PRINT #1, "define L0002 rotate(<1,0,0>,<0,0,br+360*2/5>)"
PRINT #1, "define L0003 rotate(<1,0,0>,<0,0,br+360*3/5>)"
PRINT #1, "define L0004 rotate(<1,0,0>,<0,0,br+360*4/5>)"
PRINT #1, "define L0005 rotate(<1,0,0>,<0,0,br+360*5/5>)"
PRINT #1,
FOR a = 0 TO 5
IF a > 0 THEN
   sb = 1
ELSE
   sb = 0
END IF
FOR b = sb TO 5
FOR c = 1 TO 5
FOR d = 1 TO 5
v1 = "L" + CHR$(48 + a) + CHR$(48 + b) + CHR$(48 + c) + CHR$(48 + d)
v2$ = "L0" + CHR$(48 + a) + CHR$(48 + b) + CHR$(48 + c)
PRINT #1, USING "define \ \ rotate((\ \^{*}0.5 + <0.5,0,-
0.25>),<0,0,br+360*#/5>)"; v1$, v2$, d
NEXT d
PRINT #1,
NEXT c
PRINT #1,
NEXT b
PRINT #1,
NEXT a
PRINT #1, "object { sphere L0001, 0.12 reflective_lightsteelblue }"
PRINT #1, "object { sphere L0002, 0.12 reflective_lightsteelblue }"
PRINT #1, "object { sphere L0003, 0.12 reflective_lightsteelblue }"
PRINT #1, "object { sphere L0004, 0.12 reflective_lightsteelblue }"
PRINT #1, "object { sphere L0005, 0.12 reflective_lightsteelblue }"
PRINT #1,
t$(2) = "0.1 reflective_navy"
t$(3) = "0.08 reflective yellow"
t$(4) = "0.06 reflective_coral"
n = 2
FOR a = 0 TO 5
IF a > 0 THEN
   sb = 1
ELSE
   sb = 0
END IF
FOR b = sb TO 5
```

```
continued from previous page
FOR c = 1 TO 5
FOR d = 1 TO 5
v1$ = "L" + CHR$(48 + a) + CHR$(48 + b) + CHR$(48 + c) + CHR$(48 + d)
IF b > 0 THEN n = 3
IF a > 0 THEN n = 4
PRINT #1, "object { sphere "; v1$; ", "; t$(n); " }"
NEXT d
PRINT #1,
NEXT c
PRINT #1,
NEXT c
PRINT #1,
NEXT b
PRINT #1,
NEXT a
CLOSE #1
```

This code creates two basic blocks of data. The first defines time variable positions for the elements:

```
define L0001    rotate(<1,0,0>,<0,0,br+360*1/5>)
define L0002    rotate(<1,0,0>,<0,0,br+360*2/5>)
define L0003    rotate(<1,0,0>,<0,0,br+360*3/5>)
define L0004    rotate(<1,0,0>,<0,0,br+360*4/5>)
define L0005    rotate(<1,0,0>,<0,0,br+360*4/5>)

define L0011    rotate((L0001*0.5 + <0.5,0,-0.25>),<0,0,br+360*1/5>)
define L0012    rotate((L0001*0.5 + <0.5,0,-0.25>),<0,0,br+360*2/5>)
define L0013    rotate((L0001*0.5 + <0.5,0,-0.25>),<0,0,br+360*3/5>)
define L0014    rotate((L0001*0.5 + <0.5,0,-0.25>),<0,0,br+360*3/5>)
define L0015    rotate((L0001*0.5 + <0.5,0,-0.25>),<0,0,br+360*4/5>)
define L0021    rotate((L0002*0.5 + <0.5,0,-0.25>),<0,0,br+360*1/5>)
define L0022    rotate((L0002*0.5 + <0.5,0,-0.25>),<0,0,br+360*1/5>)
define L0023    rotate((L0002*0.5 + <0.5,0,-0.25>),<0,0,br+360*3/5>)
define L0024    rotate((L0002*0.5 + <0.5,0,-0.25>),<0,0,br+360*4/5>)
define L0025    rotate((L0002*0.5 + <0.5,0,-0.25>),<0,0,br+360*4/5>)
define L0025    rotate((L0002*0.5 + <0.5,0,-0.25>),<0,0,br+360*4/5>)
```

The second block places spheres at the positions specified by these vectors:

```
object { sphere L0001, 0.12 reflective_lightsteelblue }
object { sphere L0002, 0.12 reflective_lightsteelblue }
object { sphere L0003, 0.12 reflective_lightsteelblue }
object { sphere L0004, 0.12 reflective_lightsteelblue }
object { sphere L0005, 0.12 reflective_lightsteelblue }
object { sphere L0011, 0.1 reflective_navy }
object { sphere L0012, 0.1 reflective_navy }
object { sphere L0013, 0.1 reflective_navy }
object { sphere L0014, 0.1 reflective_navy }
object { sphere L0015, 0.1 reflective_navy }
```



```
object { sphere L0022, 0.1 reflective_navy }
object { sphere L0023, 0.1 reflective_navy }
object { sphere L0024, 0.1 reflective_navy }
object { sphere L0025, 0.1 reflective_navy }
```

. . .

Each recursive layer is given its own color.

How It Works

This animation contains four basic layers of pentagons, each built on the previous layer. Each layer is signified by a nonzero element in its name, as shown in Table 8-1.

Items in level	Name of level	Color
5	L0001-L0005	reflective_lightsteelblue
25	L0011-L0055	reflective_navy
125	L0111-L0555	reflective_yellow
625	L1111-L5555	reflective_coral

Table 8-1 Item count, object names, and colors for the kaleidoscope spheres

We built each layer based on the midpoint of the previous layer plus an offset, and spin each layer so that the entire figure gears in a coordinated fashion. We've placed this entire figure inside a mirrored cone whose tip moves toward and away from the camera. Another mirror behind the camera provides additional reflections inside the cone:

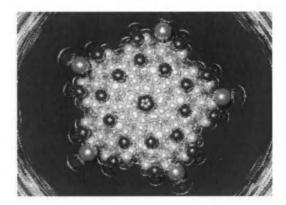
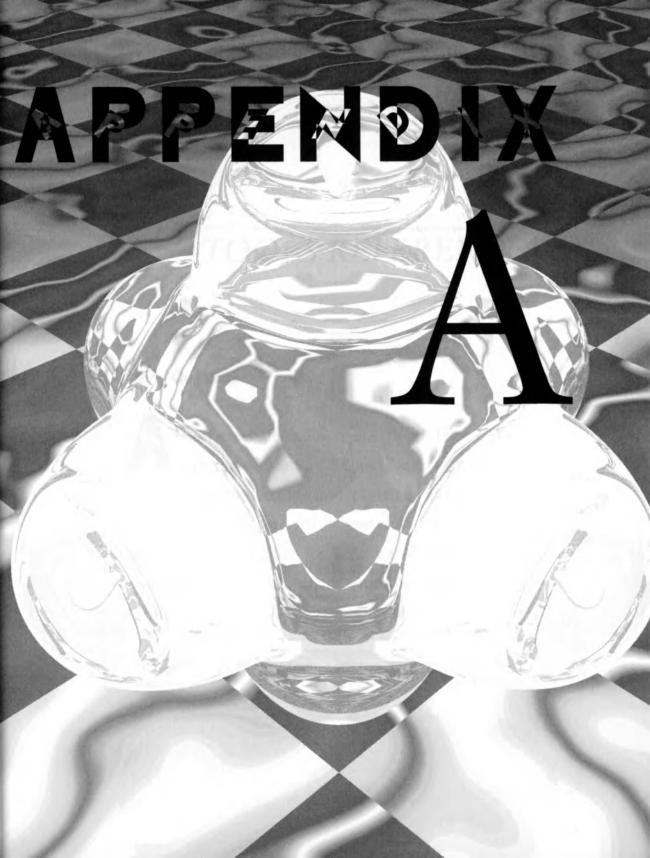


Figure 8-9 Pentagonal kaleidoscope

```
define tip -1.5+sin(radians(frame*5))
light white*0.75, <0,0,tip+0.1>
object { cone <0,0,tip>,0,<0,0,5>,10 mir2 }
object { disc <0,0,5>,<0,0,-1>,10 mir2 }
```

A combination of coordinated movement and shifting reflections makes an amazingly complex animation (see Figure 8-9) that's better than most real kaleidoscopes.





TOOLS REFERENCE

Using QuickBasic

Imost every animation created in this book is done with the assistance of QuickBasic. It's great for generating simulations and plotting mathematical functions. It's not particularly powerful or as portable as a language like C, but for what we're using it for, it doesn't have to be. It's easy to write code that runs right the first time.

Loading and Running

Even if you're new to QuickBasic, you'll be able to pick up what's being done here fairly fast. More information on the QBASIC IDE (Integrated Development Environment) can be found in the DOS documentation. The programs have already been written for you; running them is as simple as typing QBASIC (distributed with DOS) or QB, loading a .BAS file and running it. You may choose to use a mouse to move through the menus, or use the short cut commands to accomplish these actions.

The shortcut command for opening a file is (ALT)-(F), then (O), bringing up a menu that displays the files available in the current directory. You can either pick one or change to the directory where the one you want resides. Once the file is loaded, select (R)un (S)tart or use the short-cut command (SHIFT)-(F5) to run the program. The program finishes, instructs you to press any key to continue, and returns you to the text listing. To toggle back to the output screen, press the (F4) key. Believe it or not, that just about covers all you really need to know about running QuickBasic code.

Step-by-Step Example

- 1. Change to the PLY\CHAPTER2\BOLITA directory.
- 2. Enter either QBASIC or QB depending on which version you have.
- 3. Select (F)ile (O)pen, and pick COLLIDE.BAS.
- 4. Select (R)un (S)tart.
- 5. Watch the animation run. Press (ESC) when bored.
- 6. Press the spacebar to return to the text listing of the program.
- 7. Press the (F4) key twice, once to go back to the output screen, once to return to the text listing.
- 8. Exit QBASIC or QB by pressing (ALT)-(F), then (X).

That's all there is to running QuickBasic Programs.

Editors

You'll need to develop a basic understanding of the syntax, which for the most part is fairly simple. The best way to learn any programming language is by example. You see something interesting, read the code, and extract those things you find useful. This is where a good text editor, one that allows multiple windows to view several programs at once, and skill using cut and paste command come in very handy. The shareware editor Q or the expensive but wonderful editor BRIEF are two fine examples, but EDIT

(which comes with DOS) or even NOTEPAD (which comes with Windows) can suffice if that's all you have.

QuickBasic Program Structure

Most programs start by defining the screen and setting the extents and colors. Then some math is done, put on the screen, and animated. Finally, files are opened to write the information to the disk, most times Polyray data files, which generate the final images and animations. Let's go over how these operations are done.

Setting Graphics Mode

Every program displaying graphics selects a video mode and defines coordinates for the lower left and upper right corners of the screen. We'll use mode 12, which gives us a 640 x 480, 16 color screen, since all VGA's can do it, and it's fairly high resolution. Depending on the size of the objects you plan to use, the WINDOW command sets the appropriate size, but you should always keep a 4 x 3 aspect ratio between x and y, for example (1.6, 1.2), (80,60) and (320,240), otherwise images you simulate will end up being stretched or squashed in the ray tracer. An example of setting up the graphics screen follows:

```
SCREEN 12
WINDOW (-320,-240)-(320,240)
```

The default values for the colors in the VGA palette are not only random, they're unpleasant, and what's worse, there are only 16 of them. Often you'll want to use color to show depth, which only works if the colors are in some logical order, like a rainbow. Achieving a rainbow palette, with colors that progress from red to violet, is accomplished with the following code. The DATA statements hold 16 sets of intensities for red, green, and blue, one set for each palette entry, whose values range from 0 to 63. If you don't like the colors here, you can always change them to any of the 262,144 available colors by simply entering different values for red, green, and blue in the DATA statements.

```
FOR y = 1 TO 4
   FOR x = 1 TO 4
        colornum = x + ((y - 1) * 4) - 1
        READ red(colornum), green(colornum), blue(colornum)
        KOLOR = 65536 * blue(colornum) + 256 * green(colornum) + red(colornum)
        PALETTE colornum, KOLOR
        COLOR colornum
```

continued on next page

```
continued from previous page
NEXT y
```

```
' Rainbow Palette
DATA 0, 0,
DATA 32, 0, 0
DATA 42, 0,
DATA 58, 16,
DATA 63, 32,
DATA 58, 56, 0
DATA 16, 42,
DATA 0, 30, 36
DATA 0, 20, 40
DATA 0, 10, 48
DATA 0, 0, 63
DATA 20, 0, 53
DATA 23, 0, 29
DATA 19, 7, 17
DATA 50, 40, 45
```

DATA 63, 63, 63

Vectors and Arrays

Math depends on what you plan to do with it, so there's not much in the way of guidelines that can be offered here, other than try to use vectors and arrays whenever possible. QuickBasic allows user defined data structures that come in very handy for holding sets of logically connected variables. Consider the following code:

```
TYPE Vector

x AS SINGLE
y AS SINGLE
z AS SINGLE
END TYPE

DIM penta(5) AS Vector

FOR a = 0 to 360 STEP 72

x = SIN(a * rad)
y = COS(a * rad)
penta(i).x = x
penta(i).y = y
penta(i).z = 0
i = i + 1

NEXT a
```

The preceding code defines a data type called a *Vector* that holds three variables: x, y and z. Rather than using a clumsy two-dimensional array, such as penta(1,3), and then having to keep track of which number refers to

the axis and which one refers to the actual point number, *Vector* keeps it clear for us. The variable penta(3).x is the third point's x value. Arrays must be given room for storage, which is where the DIM statement comes in. The variable penta can hold five sets of three values each. If you wanted it to hold 100, you'd have to use

DIM penta(100) as Vector

Do try to make your variable names meaningful to total strangers, as you'll be one yourself about a week after you've finished writing this code and gone on to bigger and better things.

File I/O

Files are opened by specifying a text name for them and assigning a number to the stream that results. The concept of data flow as streams is picturesque; the normal input stream is the keyboard, the output stream the video monitor. Redirecting that flow to a file is as easy as opening a stream and directing its output there.

The following OPEN statement is broken up into two lines here, for clarity. It could just as easily be done with one line, but then we wouldn't get a chance to mention string variables. Variables with \$ signs at the ends are string variables. Strings are a series of characters, as opposed to numeric variables like x and frame, which are single numbers.

```
file$ = "\test\pinkfloy.d"
OPEN file$ FOR OUTPUT AS #1
x = 1
frame = 2
PRINT #1, x, frame
PRINT #1, "Hello, (hello), ((hello)), is anybody IN there?"
PRINT #1, "Just nod if you can HEAR me."
CLOSE #1
```

This opens a file called PINKFLOY.D in the test directory and subsequent PRINT #1, statements put data not up on the screen, but into this file stream. Running this program would produce the following file, PINKFLOY.D:

```
1 2
Hello, (hello), ((hello)), is anybody IN there?
Just nod if you can HEAR me.
```

Note that QuickBasic will not make the test directory for you if one doesn't already exist, but it's not shy about issuing an error message informing you of this problem. To run this, create a test directory first.

Sequential Numbering

Numbered lists come up so often that they deserve a few minutes of consideration here. Let's say you're trying to generate a sequential list of numbered variables. Variables are strings, and numbers can be converted to strings with the STR\$() function. Your first attempt is

The problem with these variables is they all have spaces in them, which isn't allowed. QuickBasic has an LTRIM\$() function that removes leading blank characters from strings, so next we try

```
For x = 1 to 15
   item$= "item"+ LTRIM$(str$(x))
   print item$
next x
```

The output of this is

```
item1
item2
item3
...
item9
item10
item11
item12
...
```

The only problem with this is that if you passed these variables to a program that needed to process these items in order, the order it would pick would be

item1 item10 item11 ... item2

In other words, it puts variable 2 after variables 11-19. This problem can be addressed by using leading zeros—as many as will allow an unambiguous order for the largest number you need to use.

Consider the following loop:

```
OPEN "test" for output as #1
pref$ = "test"
FOR frame = 0 to 15
    frame.count$ = RIGHT$("0000" + LTRIM$(STR$(frame)), 4)
    frame.pi$ = pref$ + frame.count$ + ".pi"
    frame.tga$ = pref$ + frame.count$ + ".tga"
    PRINT #1, frame.pi$
    PRINT #1, frame.tga$
NEXT frame
CLOSE #1
```

This code creates a file called TEST that contains the following text:

```
test0000.pi
test0000.tga
test0001.pi
test0001.tga
test0002.pi
...
test0009.pi
test0009.tga
test0010.pi
test0010.tga
test0011.tga
```

Consider the case where *frame* = 5. The line

```
frame.count$ = RIGHT$("0000" + LTRIM$(STR$(frame)), 4)
```

converts the value for *frame* to a string, tacks it onto the end of the string "0000", to make the five character string "00005". It then trims the result to include only the right four digits. This gives us "0005". You'll find this comes in very handy at times.

PRINT #1, USING...

Another useful construct is the PRINT #1, USING ... command, which allows you to specify the positions and lengths of variables inside PRINT expressions. A five character string expression is located in the PRINT #1, USING statement by including the structure "\ \" (a backslash, three spaces and another backslash) where you want the variable to go, and numeric expressions that specify the number of digits and the placement of the decimal point using "#.###" structures. The following example:

```
v1$ = "L0001"
v2$ = "L0100"
d = 3
```

continued on next page

Underscores

A common delimiting character in C programs, used liberally in the COLORS.INC file in Polyray, is the underscore character "_". QuickBasic interprets this as a formatting character in PRINT USING statements. The following program

In a simple PRINT statement, the underscore is passed as just another character. In the first PRINT USING statement, its use changes to mean "print the next character as a literal character." This allows for things like printing # symbols in PRINT USING statements, rather than their being treated as placeholders for numeric variables. In the second line, it prints the literal value for r, which is just r, and the underscore disappears. The third line uses two underscores, which prints the second underscore as a literal underscore, giving us what we needed.

Quotation Marks

Polyray data files require quotation marks surrounding include files such as "colors.inc".

```
include "colors.inc"
```

PRINT statements use quotation marks to signify the ends of string variables. To get a PRINT statement to generate a quotation mark, you break the strings up and insert the ASCII code for a quotation mark (CHR\$(34)) as just another component to be printed. The following print statement

```
PRINT #1, "include "; CHR$(34); "colors.inc"; CHR$(34)
```

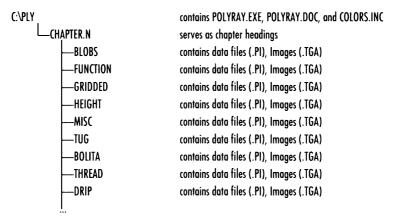
writes the line include "colors.inc"

to the output file #1.

Using Polyray

Installation

The CD contains an installation program that will automatically load the Polyray program, the data files, and all the utilities required into their proper locations. Let's cover the directory structure that results and what it does for us. The main directory for Polyray is PLY and everything is loaded as subdirectories off this one, resulting in the following organization:



The executable program POLYRAY.EXE and the Polyray documentation file POLYRAY.DOC reside in the main PLY. Any time you need access to the docs, enter:

EDIT \PLY\POLYRAY.DOC

and a full text listing of the documentation file will appear. You might also print out a copy of this, but paper can't be searched for keyword matches as easily as this electronic copy.

The include files for colors and textures commonly used in all images are placed in the PLY directory. The image creation data files, with .PI extentions, reside in subdirectories below that one, collected roughly by topic. The shareware version of Polyray includes a batch file that will create all the sample images supplied with Polyray. We didn't do that with the animations in this book because it would take a long time, and require several gigabytes of disk storage space. TGA files tend to form sprawling

mounds, and it's useful, at least from a directory listing standpoint, to herd them together into their own subdirectories, use them to create flics, then blow them away.

Running Polyray

Detailed instructions are found in the Polyray documentation which cover all the intricacies of running Polyray. However, there are a few things you really need to know about it at this stage. We'll go over the basic concepts, give a few shortcuts, and provide a step-by-step example.

Running Polyray involves going to the subdirectory containing the data file you wish to render, entering POLYRAY followed by the name of that file, which has a .PI extension, and (optionally) providing a name for the output image.

POLYRAY filename .PI -o filename .TGA

This uses the input file *filename* .PI and generates the output image file *filename* .TGA. The default output name is OUT000.TGA, and animations usually specify their own names when they are rendered.

Running Polyray from a Batch File

The preceding example assumed that the program POLYRAY.EXE was somewhere in your PATH. The executable program POLYRAY.EXE is kept in the \PLY directory. Rather than extending your path statement, you can use the batch file PR.BAT, and place that file somewhere in your PATH (like the \DOS or \TOOLS directory if you have one). The PR.BAT batch file reads

\PLY\POLYRAY %1.PI -o %1.TGA %2

Then, when you wanted to render a file called BLOBS.PI, you'd type the line:

PR BLOBS

The first parameter %1 equals BLOBS. This tells POLYRAY to look for the file BLOBS.PI and create an output file called BLOBS.TGA. The %2 parameter is included so that when you want to resume a trace you (or your pet, or the power company) paused, you can enter

PR BLOBS -R

and Polyray will pick up where it left off. If the file BLOBS is not a single image, but rather an animation, the name the batch file suggests (%1, TGA) for the output image will simply be ignored.

Step-by-Step Example

Here's a step-by-step process to follow for all this.

- 1. Copy the file PR.BAT from the PLY directory to a subdirectory in your path.
- 2. Type CD \PLY\DAT\BLOBS at the command line.
- 3. Type PR BLOBS.
- 4. Wait for the image to finish rendering and display the summary screen.

A file called BLOBS.TGA will have been created on your hard drive. It is a 24-bit targa file that requires some post-processing and image display software to view. We'll cover the next few steps in more detail later on, but to see this image immediately, use the following steps.

- 5. Enter DTA BLOBS.TGA /fg.
- 6. Enter AAPLAYHI BLOBS.GIF.

Step 5 converts the 24-bit targa file to an 8-bit .GIF file, which can be viewed using the flic player AAPLAYHI. While it may seem odd to use an animation player to view a static file, hey!, it works. Other programs like CSHOW and VPIC are more commonly used for image viewing, along with several windows programs, but we can't assume you have any of those at this point.

Include Files

The structure we've outlined is important for one other thing. Most Polyray files make use of color and texture definitions found in the files COLORS.INC and TEXTURE.INC. These are included in each Polyray data file (.PI) by the lines:

include "\PLY\COLORS.INC"
include "\PLY\TEXTURES.INC"

The directory containing these include files must be named "PLY", otherwise references made to them in each datafile would need to be changed.

POLYRAY.INI

The POLYRAY.INI file allows you to specify runtime parameters that control things you probably don't really want to have to worry about every time you run Polyray. While Polyray will run without this file, it defaults to

running without displaying the image, which makes a lot of people uncomfortable. (What's it DOING!?!)

Typically, when you start a new animation group, change to the PLY\DAT directory, make a new subdirectory off that one (say, BOUNCE), change to that directory (CD BOUNCE), then copy a default POLYRAY.INI file you keep handy in the PLY\DAT directory into \BOUNCE. The only way (currently) for Polyray to use the information in POLYRAY.INI is for it to reside in the current directory you're in when Polyray is started.

A typical POLYRAY.INI file contains the following lines:

```
abort_test
                 0.01
alias threshold
antialias
                 none
display
                  vga
                  7
max_level
                 10
max_samples
pixel_size
                  24
pixel_encoding
                 rle
renderer
                 ray_trace
shade_flags
                 63
shadow_tolerance 0.02
status
                  totals
```

The most important variables to consider now are

abort_test on display vga status totals

The abort_test on parameter tests to see if a key's been pressed and stops Polyray, writing whatever image it has created so far to the disk drive before shutting down. Checking for a keypress while Polyray is running takes a little more CPU time than running without it, but it's usually no more than a few seconds extra per trace. The only way to stop a trace with abort_test off is to reboot your computer, which is not particularly subtle. Keyboards become magnets for pets, children, companions, innocent bystanders and mounds of floppies when you leave your computer unattended rendering animations for long periods of time. Turning off abort_test is usually a good move in such cases.

The display vga parameter shows the image as it renders, but don't be disappointed with the quality. This mode is a low-resolution VGA approximation of a much higher quality 24-bit image that's actually being written to the disk. Another display option, display hicolor, is available, and if you have hicolor hardware that's VESA compatible, it will display a much better looking image as it renders. VESA compatibility is either in the cards BIOS, or may be obtained using a TSR program like UNIVERSA. Most

people have VGA or SVGA hardware. If you have a hicolor display adapter, try it out. This mode does not work with at least one 24-bit Windows accelerator card that has hicolor support, but it works with most other hicolor cards.

The *status totals* parameter shows a lot of really neat but mostly cryptic information. The two most important pieces of information are the frame you're about to render and how long it took to render the last one.

Polyray Data File Syntax

The easiest way to pick up the syntax of Polyray files is to render all the sample images that come with it and look at the code that created those images. The batch files make this process as simple as entering a single command. It could not be easier. The Polyray documentation covers all available graphics primitives and the syntax for their use. While this does not make for casual reading, questions regarding specific items of syntax and usage are thoroughly covered in the documentation, and are only a simple text search away.

All data files have a viewpoint block that sets the camera location and image size, and other items that specify light sources, and objects. Animations have animation frame counters and defined functions that generate the motions. Colors and textures are either defined or included from libraries of textures, and applied to objects. Beyond these basics, the field is wide open. Here's how it looks:

```
start_frame 0
end_frame 32

outfile ball

viewpoint {
   from <0,5,-5>
   at <0,0,0>
   up <0,1,0>
   angle 30
   resolution 32,20
   aspect 1.433
   }

include "\ply\colors.inc"
background SkyBlue

light white, <-5,10,-5>

define grow frame\10
```

continued on next page

```
continued from previous page
object {
    sphere <0,0,0>,grow
    shiny_blue
}
```

Creating Image Data Files

You should use whatever tools you're comfortable with to generate your data files. There are now many add-on utilities that allow you to graphically design scenes for your favorite ray tracer (POV Cad, Sculptura, Imagine, RTAG, Animake, MORAY, and many more exist as well) which either output compatible scene source code, or save it in some universally accessible format that can then be translated. Most CAD packages save DXF files, which are gruesome and longwinded but provide a standard format with published specifications, and translators are available to go from this standard to ray tracers.

Sometimes all you need are triangle points (x1,y1,z1,x2,y2,z2,x3,y3,z3) that specify the corners of a bunch of little triangles, which used together define some object, like a cow or Beethoven. It simply becomes a matter of converting this data, one triangle at a time, into acceptable Polyray syntax. This is done using a program, and an example is provided in this book in Section 5.4. Then anything, even objects originally defined for high-end workstations, can be translated into ray tracer compatible data.

Standard paint programs can be used to provide both imagemaps for textures and heightfield data for raised terrains and extruded logos. The ability to import 2-D data to generate 3-D objects is incredibly powerful.

For complicated files containing hundreds or perhaps thousands of items, it is easiest to handle such things programmatically. You first generate a simple file containing a couple of these items, then automate the creation of all the other ones. The ability to use programs to generate image files makes text-based ray tracers much more powerful than graphics-based image creation programs.

Commenting the Data Files

The double backslash (\\), commonly use in C++, is interpreted as "ignore this line, it's just comments." Liberal comments are always a good idea. The only time you'll completely understand your own code is while you're writing it, and later on you'll wonder what possessed you to do something a certain way. Comments rescue you from a continual journey of rediscovery. Comments also come in handy during image development, blocking out unnecessary items to simplify the source code, to make it render faster.

Most code goes through several generations before it looks good enough to keep. Commenting out old code and adding new code gives you an audit trail, allowing you to backtrack to where you were. Also, saving each revision of a file to a sequential name (SURF1.PI, then SURF2.PI, then SURF3.PI...) allows you to keep files around that might not seem important at the time, but provide invaluable clues later on. There's nothing worse than having a really great image, but not having the original data file because later revisions overwrote it.

Internal Versus External Animation Control

Really good animations come about when you motivate static scenes with programs of your own design. Two rough classes of animation are external and internal animation. External animation means you create a series of text files (one per frame) that sequentially generate each image. This is necessary when the data used for the motion is not easily incorporated into Polyray's syntax, or comes from complex programs written in different languages. Internal animation uses a single data file in which variables are controlled by a frame counter that steps the action of static objects through their motion. Internal animation is cleaner in that there are fewer files hanging around. Most of the animations in this book are internal, but examples of external animations are provided as well.

Using DTA

Polyray generates a series of images in a numbered list. For ten frame animation with an outfile specification of HAPPY, it would create the following ten files:

```
HAPPY000.TGA
HAPPY001.TGA
HAPPY002.TGA
HAPPY003.TGA
HAPPY004.TGA
HAPPY005.TGA
HAPPY006.TGA
HAPPY007.TGA
HAPPY008.TGA
HAPPY009.TGA
```

Creating a flic file called HAPPY.FLI from this series might involve entering the following line:

DTA HAPPY*.TGA /oHAPPY /s5

You hand DTA the file prefix for the images, give it an output name HAPPY with /oHAPPY, and tell it to have a playing speed of 5 with /s5. DTA defaults to flic output.

There are many switches in DTA, and please read the documentation for more information on them. They're all very useful, but two in particular are used all the time. If you generate images for your flics that are larger than 320 x 200, you must use the /R6 switch to make a high resolution .FLC from them. There are other modes, so again, read the documentation. A quick summary is also available if you just enter the command DTA by itself with no other parameters.

One of the first tasks DTA must perform is generating an optimal palette to map the 16.7 million possible colors in the TGA files into the best 256 colors to run on a standard VGA or SVGA display. This can be a lengthy process for animations that are several hundred frames long, and if the basic image doesn't change all that much, checking every single TGA file doesn't really make higher quality flic. You can scan, say, every 10th image using the /c10 switch, and this can really cut down the time it takes to make the final flic.

DTA generates GIFs using the /fg switch. As mentioned earlier, the animation player AAPLAYHI can then display these GIFs for you, although VPIC or CSHOW, two popular shareware GIF viewers, are more commonly used for this task. CSHOW and VPIC can also display TGA files directly.

Step-by-Step Example

Here's a step-by-step process to follow for all this:

- 1. Change to the the PLY\CHAPTER1\TEST directory.
- 2. Run the file TESTANIM.PI with Polyray by entering PR TESTANIM.
- 3. Wait for the ten frames to render.
- 4. Generate a flic by entering DTA test*.tga /ff /otest /s5.
- 5. View the flic by entering AAPLAYHI test.fli.
- 6. Press (ESC) when you're done watching it.

Using AAPLAYHI

AAPLAYHI is a freely distributable animation player program from Autodesk. The default maximum display size is 320 x 200, but with the

appropriate video cards and VESA compatible modes, animations as large as $1,024 \times 768$ can be displayed. It not only works with flics, but will also display GIFs as well.

An AA.CFG file is created every time AAPLAYHI is started from a directory that doesn't contain one, and the screen size may be selected from whatever screen modes AAPLAYHI detects your hardware is capable of.

If your flic is 320 x 200 or smaller, you can run AAPLAYHI directly by entering the command AAPLAYHI *flic*.FLI. Entering AAPLAYHI by itself brings up a graphics screen and identifies the program. You press a mouse key to get to the menus. At this point, you'll either be able to load your flics directly or set the screen size to the appropriate one and load your flics afterwords.

AAPLAYHI will use all your system memory and attempt to load the flic into it, which makes flics play much smoother than directly off your hard drive. While it doesn't mind HIMEM, you cannot have EMM386 providing expanded memory or AAPLAYHI will key on it and miss your extended memory entirely.

Step-by-Step Example

Here's a step-by-step process to follow for all this.

- 1. Change to the the PLY\CHAPTER1\TEST directory.
- 2. Enter AAPLAYHI by itself.
- 3. Press a mouse key to clear the screen.
- 4. Go to the menu and select Flic, Load, and Test.
- 5. Press the >> button and watch the flic play.
- 6. Press any key to pause.
- 7. Select Program, Quit, and exit AAPLAYHI.

ANIMANIA



MOVIE CONTEST



Waite Group Press is going Hollywood! Here's your chance to show off your knowledge of computer animation and dazzle us by using the programs in this book to create you own movie. You don't need a studio-sized budget, just your creativity, a spectucular viewpoint, and a little spirit of competition. You could quickly be on the road to riches and fame.

Here's what you can win:

- First place winner receives \$1000
- Second place winner receives \$500
- Third place winner receices the entire Waite Group Press library
- Fourth through tenth place winners receive Waite Group Press T-shirts

To enter, mail your *original flic* and source code, with the entry card on the back of this page or a 3x5 postcard that includes your name, address, daytime phone number, the title of your entry, and a brief description of the movie.

Rules and Regulations

Eligibility: Your submission must be submitted between February 1, 1994 and August 31, 1994; it must be created with the programs provided in this book.

Dates and deadlines: Entries must be received at the address provided by 5PM, August 31, 1994. Winners will be notified by February 28, 1995.

Preparation of entries: Only one flic may be submitted per entry, but you may enter as often as you like. Entries must be on a 3.5-inch disk and each disk must be labeled with your name, phone number, and the name of the flic. Waite Group Press is not responsible for lost or damaged entries. Waite Group Press will not return any entries unless entrants include a self-addressed stamped envelope designated for that purpose with their entry.

Media: Entries may be submitted on tape or disk. Entries submitted on tape should be sent in either QIC-80 or DAT compatible formats. Entries submitted on disk should be archived with the DOS backup command or as a self-extracting archive.

Judging and prizes: Flics will be judged on the basis of technique, creativity, and presentation by a panel of judges from Waite Group Press. Winners will be notified by February 28, 1995. For the names of the "Animania Movie Contest" winners and their entries, send a self-addressed, stamped envelope, after March 15, 1995, to: Waite Group Press "Animania Movie Contest" Winner, 200 Tamal Plaza, Suite 101, Corte Madera, CA 94925.

Etc: No purchase necessary. Waite Group Press "Animania Movie Contest" is subject to all federal, state, local, and provincial laws and regulations and is void where prohibited. Waite Group Press "Animania Movie Contest" is not open to employees of The Waite Group, Inc., Publishers Group West, participating promotional agencies, any WGP distributors, and the families of any of the above. By entering the "Animania Movie Contest," each contestant warrants that he or she is the original author of the work submitted, non-exclusively licenses The Waite Group, Inc. to duplicate, modify, and distribute the flic, and agrees to be bound by the above rules.

		Daytime Pho	ne
City		State	Zip
Entry (file name)			
	now about this flic?		
Send entries to:	"Animania Movie Contest"		
Send entries to:	"Animania Movie Contest" Waite Group Press		
Send entries to:	"Animania Movie Contest" Waite Group Press 200 Tamal Plaza		

INDEX

1.40	
/c10 switch	animation examples
Dave's Targa Animator (DTA), 466	ASTEROID, 393-403
// (comments)	BANARAMA, 373-380
Polyray, 84-85, 464-465	BARCODE, 114-119
/fg switch	BOG, 156-160
Dave's Targa Animator (DTA), 466	BOLITA, 58-76, 257
/R6 switch	BORG, 423-428
Dave's Targa Animator (DTA), 6, 466	BOX, 240-245
\Leftarrow symbol, xix	BROWNIAN, 301-304
	BUCKY, 312-322
A	BUMPERS, 256-260
AA.CFG files, 467	BUTTER, 429-443
AAPLAYHI animation player	CARB, 206-211
converting TGA files to GIF, 461	CELL, 166-172
getting started, xviii-xix	COIN, 14-25
overview, xiv	DRIP, 77-89
reference, 466-467	FAKE, 147-156
abort_test parameter	FAN, 409-413
Polyray, 462	FLICKRPOT, 160-166
Animake program, 464	HMM, 215-225
Animania Movie Contest, 468-469	IMPACT, 182-188
animation	ITERBOX, 413-423
external, 321-322, 465	JELLYDIM, 188-202
internal, 465	KALE1, 443-448

animation examples (continued)	В
MORAVIAN, 202-206	backgrounds
MORAY, 464	flickering, 89
NTCLUB, 129-134	backslashes (\\), 464
REDLIGHT, 260-266	backup systems, <i>xv-xvi</i>
RINGS, 322-331	ballistic effects, 269
ROCKET, 47-58	balls
ROCKY, 119-128	
RTAG, 464	on flipping coins, 20-22
SHATBLOB, 274-279	basics, 3-25 batch files
SHATTER, 267-273	
SPHERE, 331-336	BBANIM.BAT, 320, 322
STARS, 380-385	DOIT.BAT, 442-443
SWARM, 292-301	generating animations with, 161
SWOOP, 30-47	information in, 322
THREAD, 230-240	ORBITER.BAT, 400
3LEVELS, 363-372	PR.BAT, 145, 460-461
TILE, 90-96	program to create, 162-163
TUG, 279-291	RUNEM.BAT, 145
TUMBLE, 7-14, 97-109	running Polyray from, 460-461
TUNL, 134-147	Bezier patches, 160-162, 230-232, 239-240
TWINKLE, 350-358	blobs, 173-175, 274-279, 348, 361-403
UDDER, 386-390	BRIEF text editor, xiv, 452
VOLCANO, 304-308	Brown, Truman, 348
WATRTUNL, 390-392	Brownian function, 301-304
WATUSI, 226-230	buckyballs
animation files, xiv	tumbling, 312-322
animations	bug eyes, 350
postage stamp, 7	
running in batch mode, 322	C
steps in creating, 18	C-60 buckyballs
antialiasing, 5, 46	tumbling, 312-322
ARJ files, 301	CALL command, 145
Arrequin, Jorge, 59	cameras
ASCII code	exterior shots, 151
quotation marks, 458	motion
aspect ratio, 6, 453	around objects, 109
audit trails	deltoid, 91, 93
using comments for, 465	happy train, 216
Autodesk 3D Studio program, xiii, 211	linear, 30, 47
- -	* *

cameras (continued)	command-line syntax (continued)
motion (continued)	Polyray, 460-461
orbital, 30, 109	comments
rising and falling, 91-93	in Polyray syntax, 464-465
spiraling, 33-34	saving old values in, 84-85
with texture changes, 119-128	compressed files, xvi, 301
through tunnels, 47-58, 71-76, 134-	cone clipping, 211, 214
147, 392	Connect the Dots (CTDS) utility, 348
tumbling, 172	constructive solid geometry (CSG), 105-106
positioning, 36-37	contest
shifting viewpoints of, 113-177	Animania Movie Contest, 468-469
speed control of, 34	coplanar triangles, 190, 194-195
spline paths for, 134-147	Corel Paint program, xvi
visible paths for, 38-42	crystals
"Cannot allocate transformation"	creating and distorting, 279-291
message, 424	CSHOW GIF viewer, 461
Cartesian coordinates, 307	cubes
center rotation, 229-230	recursive, 418-423
chaos maps, 313	rubbery, 240-245
chaser lights, 52-57	shattering, 267-273
checkerboards	tumbling, 9-14, 97-109, 134
overuse of, 91	curves. See also splines
CHR\$(34)	bell-shaped, 35-36
quotation marks, 458	
CLS (clearscreen) function, 8	D
coins	_
flipping, 14-25	DATA statement
collision detection, 256-260	QuickBasic, 453-454
color banding, 6, 114-119, 128	data storage devices, xv-xvi
colors	data structures
definition files for, 461	QuickBasic, 454-455
extending range of, 6	Dave's Targa Animator (DTA)
mapping, 121-124, 128-129	/c10 switch, 466
Maxfield Parrish-type, 356-357	/R6 switch, 6, 466
primary, 331	assembling final flic, 300-301
RGB specification for, 118	getting started, xviii
VGA palette, 453	@list feature, 64
command-line syntax	list mode, 300
AAPLAYHI animation player, 466-467	overview, xiv
Dave's Targa Animator (DTA), 465-466	post processing with, 25

Dave's Targa Animator (DTA) (continued)	error messages, 424
reference, 465-466	expanded memory
decision spheres, 166-167, 171-172	and AAPLAYHI, 467
delimiters	extended memory
underscore character as, 458	and AAPLAYHI, 467
deltoid	external animation, 321-322, 465
camera motion, 91, 93	
shape, 330	
viewpoint, 331	F
Deren, Eric, 267, 274, 292, 355, 414	
DESKTOY animation, 59	"Failed to allocate CSG node" message, 424
DIM statement	Farmer, Dan, 301
QuickBasic, 455	files. See also animation examples; Polyray
directories	data files; QuickBasic files
multiple linked, 299-300	AA.CFG, 467
Polyray, 459	ARJ, 301
display hicolor parameter	compressed, xvi, 301
Polyray, 462-463	DXF, 464
display rate, 5	flic, 300-301
display vga parameter	GIF, <i>xiv</i> , 461
Polyray, 462	INC, 42, 459
displays	names in batch files, 322
undrawing, 8	PI, 42, 459
dithering, 6, 160	TGA, xiv-xv, 459-461
DKB-Trace ray tracer, 337	fireworks, 256
DXF files, 464	flags
	shading, 134
	flicker, 8, 89, 164-165
E	flics
	described, xiv
EDIT text editor, xiv, 452-453	files, 300-301
editors	players, 4
text, xiv, 452-453	size, xv , 5
using text, 123	Floyd Steinberg dithering, 6
electrostatic repulsion, 282	fountains, 256
EMM386	Fractal Programming in C, 409
and AAPLAYHI, 467	fractals
Enzmann, Alexander, 4, 274, 380	L-system, 408
equations	frame count, 5, 114
bubbly surfaces, 156-157	frame counter, 15-16, 22-23, 269, 322, 465
rotating objects, 11-12, 97-98	frame variable, 15-16, 30

Fuller, Buckminister, 312	image size, 5
functional objects, 274	Imagine program, 464
	INC (include) files, 42, 459
•	include commands, 126-127
G	input data stream
Garcia, Oscar, 136	QuickBasic, 455
gas dynamics, 256	interface
Gaussian functions, 34-36, 45, 65, 182, 188	graphical user, xii
GIF files, xiv, 461	internal animation, 465
glides	IPAS routines, 211
spiraling, 32-45	
gridded objects, 94-95	J
GRIDMASK.TGA, 94-95	
	jaggies, 5
Н	K
Hammerton, John, 337, 343, 386	KLICKL animation, 59
Hanrahan, Pat, 129, 206	,
hard drives, xv	
hardware	L
hicolor, 462-463	layered recursion, 413-423
limitations, 5	Leech, Jon, 355
requirements, xv , 423-424	LIFE cellular automata program, 255
heightfields	lighting
complex surfaces, 31	chaser lights, 52-57
data from paint programs, 464	orbital, 281-291
described, 31, 182	shifting patterns, 129-134
specifying detail, 46, 159-160	spotlights, 85, 118-119, 129
hicolor hardware, 462-463	line overflow symbol (\Leftarrow), xix
hicolor VGA, xvi	linear motion, 30, 47
HIMEM	@list feature
and AAPLAYHI, 467	Dave's Targa Animator (DTA), 64
	list mode
•	Dave's Targa Animator (DTA), 300
	lists
IBM logo, 215-216	numbered, 465, 546-457
image creation data files (.PI), 42, 459	logos, 215-225
image definition code, 407	capturing from television screen, 337
image files	data from paint programs, 464
names of, 322	loop points, 4, 256

looping, 4, 64, 146, 418 looping splines, 135 L-system fractal tree, 408 LTRIM\$() function QuickBasic, 456-457 LZW files, 301	motion (continued) recursive, 409 rising and falling, 91-93 start-stop-start, 260-266 through tunnels, 47-58, 71-76, 134-147, 392 tumbling, 9-14, 97-109, 134 waveform, 293
M	MS-DOS, <i>xiv</i>
Making Movies on Your PC, xix, 135 Mason, David, 7, 59, 135, 337 math corresponding	N
math coprocessor, xv McNair, Harold M., 216 memory and AAPLAYHI, 467 recursive objects and, 423-424 requirements, xv metaballs, 77, 361-403, 429-442 mirrors, 147-156, 329, 443-448	near warps, 321 noise dithering, 6 noise surface textures, 120-122 NOTEPAD text editor, 453 NTSC video output card, xvi numbered lists, 465, 546 numbers
Mitchell, Don, 129, 206 Monte Carlo simulations, 256 morphing	dimensionless, 22 numeric variables QuickBasic, 455-456
disco inferno night club, 206-211 moose, 175-177 Moravian stars, 202-206 pat of butter, 429-443 three-dimensional, 188 3-D reference points, 350 motion	objects defining, 407 functional, 274 gridded, 94-95
ballistic effects, 269 coordinating collections of objects, 311-358 curves, 35-36, 134-135, 393-396 fluids, 77-89 hypnotic, 292-301 layering, 24, 429 linear, 30, 47 orbital, 30, 109 periodic, 58-76 phased, 189, 194	internal structure of, 363 triangle-based, 274 OPEN statement QuickBasic, 455 orbital motion, 30, 109 Otwell, Douglas, 337 outfile names modifying, 300 output data stream QuickBasic, 455 overflow symbol (⇐), xix overview, xi-xiv

P	Polyray (continued)
paint programs, 464	internal animation, 465
Palczewski, Hank, 59	overview, xiv
parameters	parameters, 460-463
Polyray, 460-463	parser, 362
Parrish, Maxfield, 356-357	recursive images in, 408-409
particle systems, 255-308	reference, 459-466
collision detection, 260-266	commenting data files, 464-465
random diffusion, 267-273	controlling animation, 465
tracking motion in, 304-306	creating image data files, 464
paths	data file syntax, 463-464
oval, 93	include files, 461
spiral, 32-45, 231	installing, 459-460
spline, 134-135, 231, 256, 393-396	POLYRAY.INI, 461-465
visible, 98	running, 460-461
periodic motion, 58-76	scalars, 239
Photoshop program, xvi	solver code, 380
physical recursion, 409	translating from QuickBasic code, 13-14,
PI files. <i>See also</i> Polyray data files	46
PI (image data) files, 42, 459	triangle output option, 274
Picture Publisher program, <i>xvi</i>	variables, 238-239
pixel shape, 6	vectors, 171, 239
pizza box routine, 210-211	wire frame rendering mode, 7
PLY Polyray directory, 459	Polyray data files
polar coordinates, 307, 351	ANGRY.PI, 85-89
polygons	ANIM.INC, 396, 400
interlocking, 322-323	ASTEROID.PI, 396
Polyray	ASTR2.PI, 400-403
affect of parameters on surfaces, 385	BANA.PI, 378-380
collision detection, 263	BARCODE.PI, 114-119
creating random numbers, 301-304	BBS.INC, 320
data files	BLOB.RAW, 274
syntax, 463-464	BOG.PI, 158-160
viewpoint block example, 463-464	BOLITA.PI, 61-65
directory structure, 459	BOLITA2.PI, 67-71
documentation, 459	BOLITA3.PI, 71-76
error trapping, 35	BOX.FLI, 245-247
external animation, 321-322, 465	BOX2.FLI, 248-251
generating code fragments, 123-124	BROWNIAN.PI, 301-304
getting started, <i>xvii-xviii</i>	BUCKY.PI, 320-322
getting started, xvii-xviii	

Polyray data files (continued) Polyray data files (continued) Butter Animation, 433-434, 440-442 RINGER.PI, 327-331 CARB.PI, 207-211 RINGZ.PI, 330-331 COIN.PI, 18-25 ROCK1.PI, 49-51, 121, 124-126 COL.INC, 267-268, 270 ROCK2.PI, 126-128 COLORS.INC, 458, 461 ROCKET.PI, 54-58 CUBIC.INC, 136-137, 141 SHATBLOB.PI, 276-279 DIAM1.PI, 195-199 SHATTER.PI, 271-273 DIAM2.PI, 200-202 SPHERE.PI, 334-336 DICE.PI, 103-106 Spiraling Bezier Animation, 234-236 SPLINE.3D, 136-137 DICE3.PI, 107-109 DOME.PI, 356-358 SQUISH, 362 5DRIP.PI, 77-81 STAR.PI, 203-206 5DRIP1.PI, 81-85 STARS.PI, 380-385 FAKE.PI, 151-156 SWARM.PI, 294, 297-300 Flickering Teapot, 165-166 SWOOP.PI, 42-46 Fractal Fan Animation, 413 TEAPOT.INC, 162 FRAGBLOB.INC, 279 TEXTURE.INC, 461 HEXCELL1.PI, 167-171 THREAD.INC, 236 HM.INC, 216, 220-222, 226 TILE.PI, 95-96 HMM.PI, 222-225 TILE1.PI, 91-94 IMPACT, 185-188 TUG.INC, 281-282 ITER*.PI, 424-428 TUG1.PI, 282-285 ITERATE.PI, 418-423 TUG2.PI, 285-287 KALEI.INC, 445-447 TUG3.PI, 287-291 KALE1.PI, 443-444 TUMBLE, 12-14 3LEVEL1.PI, 364-365 TUNL09.PI, 142-145 3LEVEL2.PI, 366-369 UDDERNON.PI, 387-390 3LEVEL3.PI, 369-372 Volcano Animation, 306-308 MARKER.INC, 98, 104, 106 WATRTUNL.PI, 390-392 WATUSI.PI, 226-230 MOOSE.PI, 175-177 MTV.PI, 345-350 POLYRAY.DOC, 459 **POLYRAY.EXE**, 459-460 MWAVE.PI, 211-214 NTCLUB.PI, 130-134 POLYRAY.INI file, 7, 24, 156, 461-465 Orbitals, 264-266 ponging, 386 PARTS.INC, 162 POP, 172-177 PARTZ.INC, 163-164, 166 postage stamp animations, 7 POP.PI, 173-175 POV Cad program, 464

POV-Ray ray tracer, xii, 337, 414

RAINBOW.PI, 120-122

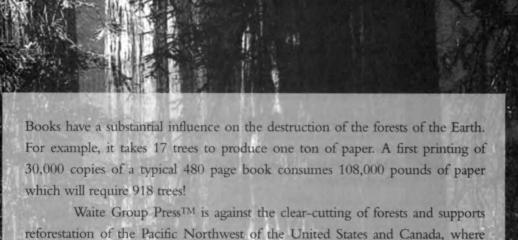
REAL.PI, 148-151

PRINT commands	QuickBasic (continued)
QuickBasic, 123-124, 146, 238-239,	reference (continued)
306, 457-459	PRINT commands, 457-459
	program structure, 453
	quotation marks surrounding file
Q	names, 458-459
Q text editor, 452	setting graphics mode, 453-454
QBASIC IDE (Integrated Development	underscore in variable names, 458-459
Environment), 452	vectors, 454-455
QEDIT text editor, xiv	STR\$() function, 456-457
QIC-80 tape cartridges, xv-xvi	string variables, 455-456
QuickBasic	translating to Polyray code, 13-14, 46
CALL command, 145	variables, 455-456
collision detection, 263	WINDOW command, 453
DATA statement, 453-454	QuickBasic files
data stream, 455	Bananarama Animation, 374-378
DIM statement, 455	BB6PI.BAS, 313-320
error trapping, 35	BEZIER.BAS, 232-234
file I/O example, 455	BOLITA, 59-61
file names	BOX.BAS, 242-245
inserting leading zeros in, 456-457	BUMPERS.BAS, 257-260
suppressing leading blanks in, 456	BUMPZ.BAS, 260-263
getting started, xvi-xvii	BUTTER.BAS, 430-433
LTRIM\$() function, 456-457	BUTTER2.BAS, 434-439
numeric variables, 455-456	COLORMAP.BAS, 122-123, 126
OPEN statement, 455	CUBEREAD.BAS, 141-142
overview, xiv	FAN.BAS, 410-413
PRINT commands, 123-124, 146, 238-	FLAP.BAS, 65-67
239, 306, 457-459	HEXCELL2.BAS, 172
rainbow palette, 453-454	HM.BAS, 216-219, 226
reasons for using, 4	JELLYDIM.BAS, 190-194
recursion in, 409	Kaleidoscope Animation, 445-446
reference, 451-467	MTV Logo Animation, 338-345
arrays, 454-455	PATH.BAS, 33-34, 39-42, 137-141
data structures, 454-455	PATH3.BAS, 396-400
editing files, 452-453	RANDOM.BAS, 161-162
file I/O, 455	RANDOME.BAS, 350-353
loading and running QuickBasic files,	RECBOX.BAS, 414-418, 423
452	RINGER.BAS, 323-326
numbered lists, 456-457	ROCKET1.BAS, 48-49

QuickBasic files (continued)	references (continued)
ROCKETZ.BAS, 52-54	software. See also AAPLAYHI; Dave's
Salt Crystal Animation, 291	Targa Animator (DTA);
SEESWARM.BAS, 293-294	Polyray; QuickBasic
SEQUENCE.BAS, 162-164	reflection, 89
Shattering Blob, 274-276	refraction, 89
Shattering Cubes, 269-270	render time, 4-5, 89, 151, 279, 363, 392
SHIFTY.BAS, 122-124, 126	repulsion
SPH2.BAS, 353-356	electrostatic, 282
SPHE.BAS, 332-334	resolution
SPLASH.BAS, 183-185	screen, 5-6
STRETCH.BAS, 145-146	resume trace parameter (-R)
SWARM.BAS, 294-297, 299	Polyray, 460
TERRY.BAS, 31-32	RGB color specifications, 118
THREAD.BAS, 236-238	rotation
TUMBLE.BAS, 9-12, 98-103	about the center of objects, 229-230
Volcano Animation, 304-306	calculations for, 11-12, 97-98
quotation marks	versus viewpoint, 106-109
surrounding file names, 458-459	runtime parameters
	Polyray, 461-463
R	_
rainbow palette, 453-454	S
RAM	scalar function, 93-94
recursive objects and, 423-424	scalar numbers, 301
system requirements, xv	scalar variables
ramp function	Polyray, 239
three-phased, 106-107	scenes
random number generator, 301	mathematical representation of, xiii
random numbers	simulating, 7-14
generating, 350-353	Schumacher, Jay, 203
Rayshade ray tracer, xii	screen coordinates, 221-222, 453
recursion, 407-448	screen flicker, 8
with fractals, 409-413	screen saver programs
layered, 413-423	Windows, 292
morphing, 429-443	Sculptura program, 464
redirection operators (DOS), 322	shadows
Reedy, Doug, 103	using shading flags for, 134
references	shapes
additional, <i>xix</i>	changing three-dimensional, 181-251

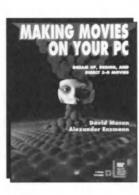
shapes (continued)	T		
pixel, 6 SIGGRAPH Proceedings (1992), 129, 206-207	tape backup, <i>xv-xvi</i> tearing, 5 terrain, 31-32, 464. <i>See also</i> heightfield		
sliced bread font, 217	text editors, <i>xiv</i> , 452-453		
Smotherman, Chris, 136	TGA (targa) files, <i>xiv-xv</i> , 459-461		
sombrero function, 156-157	3DV wire frame viewer, 136-137		
spackle, 166-167, 171	through splines, 135		
spheres	through warps, 231		
ballistic effects, 269	time shifting, 24-25		
conversion to triangle-based objects, 274	Topas program, xiii		
covered with points, 350-358	trace		
decision, 166-167, 171-172	resume, 460		
on flipping coins, 20-22	stop, 462		
tumbling colored, 331-336	treadmilling, 47, 52		
splash radius, 182-185	triangle-based objects, 274		
splines, 134-135, 231, 256, 393-396	triangle grids, 90-96, 188		
spotlights, 85, 118-119, 129	tricornered function. <i>See</i> deltoid		
SPPATH (aka SP) utility, 135	tunnels		
static keyword, 45	endless, 147-156		
status totals parameter	flight tube for rocket, 47-58		
Polyray, 463	fluid, 390-392		
Stevens, Roger, 409	inside spheres, 124-126		
STR\$() function	morae spheres, 12 i 120		
QuickBasic, 456-457	U		
string variables	U		
Polyray, 238-239	underscore characters		
QuickBasic, 455-456	delimiter, 458		
surface height, 182-183, 185	UNIVERSA program, 462		
surface retriangulation, 350	Utah Teapot, 160-161		
surfaces. See also heightfields; terrain	utilities		
bubbling mud-bog, 156-160	add-on, 464		
complex, 31, 90-96			
increasing detail of, 46	V		
switches	values		
/c10, 466	saving old in comments, 84-85		
/fg, 466	variables		
/R6, 6, 466	QuickBasic, 455-456		
	Quickbasic, 700-700		

```
vectors
  Polyray, 171, 239
VESA compatibility, 462, 467
video cards, xvi
video modes, 453
video tapes
  NTSC video output card, xvi
viewpoint block
  Polyray data files, 463
viewpoints
  versus rotations, 106-109
  shifting, 113-177
Vivid ray tracer, xii
volume rendering, 363
VPIC GIF viewer, 461
W
warps, 231
Weller, Karl, 323, 326, 331
WINDOW command
  QuickBasic, 453
Windows
  image editing programs, xvi
  screen saver programs, 292
  video accelerator cards, xvi, 463
wire frame rendering mode, 7
wire frame simulation, 189
wire frame viewers
  pseudo code loop, 8
  3DV program, 136-137
  tumbling cube example, 9-12
Y
Yost Group, 211
Z
ZIP files, 301
```



most of this paper comes from. As a publisher with several hundred thousand books sold each year, we feel an obligation to give back to the planet. We will therefore support and contribute a percentage of our proceeds to organizations

which seek to preserve the forests of planet Earth.



MAKING MOVIES ON YOUR PC

David Mason and Alexander Enzmann

Flex your imagination and direct animated movies! You'll get everything you need in this book/disk package to create fantastic rotating television logos, MTV-style action-clips, eyeboggling flybys and walkthroughs, or just about any movie effect you can dream up. The disks include the POLYRAY ray tracer for creating photorealistic images, and DTA, Dave's Targa Animator, the standard for converting ray-traced images to FLI movies. You'll also get ready-to-run example movies and explanations. No need to draw precise locations of objects and shadows—the included programs make realistic animation a smap; programming skills aren't required.

ISBN 1-878739-41-7, 200 pages, 2-5.25" disks, \$34.95. Available Now

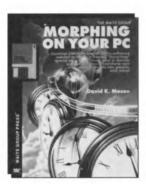


MULTIMEDIA CREATIONS

Philip Shaddock

Jump into multimedia with *Multimedia Creations* and its powerful built-in GRASP program and utilities. Whether novice or professional, you'll get everything you need to create your own electronic brochures, interactive educational games, multimedia kiosks, digital real-time movies, and music videos. Hands-on step-by-step examples teach you animation, color-cycling, how to manipulate color palettes, synchronize sound effects and MIDI music to your presentations, and much more. The accompanying disks provide fully commented examples that you can run, modify, or incorporate into your own programs.

ISBN 1-878739-26-3, 450 pages, 2-5.25" disks \$29.95, Available Now



MORPHING ON YOUR PC

David K. Mason

Change a child into a dinosaur or turn your old jalopy into a flashy sportscar. You don't need a magic wand, just this book and disk package. *Morphing On Your PC* is a complete guide to "morphing"—the digital effect that merges two images into one with uncanny realism. It's an effect used in movies, music videos, television shows, and ads, and now you can create them on your PC in six simple steps. Everything you need is provided. Try *Morphing On Your PC* and make a change for the better.

ISBN 1-878739-53-0, 198 pages, 2-3.5" disk \$34.95 Available Now

Send for our unique catalog to get more information about these books, as well as our outstanding and award-winning titles, including:

Virtual Reality Creations: Build virtual medieval or old west worlds, and walkthroughs of alien landscapes with this book and the accompanying REND386 software and Fresnel viewers.

Walkthroughs and Flybys CD: Fly around buildings before they exist, tour the inner workings of imaginary machines, and play electronic music shile watching the motion of atoms. 538MB of breathtaking computer animation and music.

Fractals for Windows: Create new fractals and control over 85 different fractal types with a zoom box, menus, and a mouse! Bundled with WinFract, 3-D glasses, world-class fractal recipes, and source code, as well as stunning color photos.

Flights of Fantasy: Programming 3-D Video Games in C++: Learn to use 3-D animation to make commercial quality video games. Includes a complete flight simulator program and the full source code.

Visual Basic How-To, Second Edition and Visual Basic SuperBible, Second Edition: Both books cover VB3, the Microsoft language that makes Windows programming more accessible. How-To provides solution to VB programming question. SuperBible is the ultimate reference on VB.

Lafore's Windows Programming Made Easy: Top instructor Robert Lafore cuts to the heart of Windows programming. Using Borland/Turbo C++. Simple C skills and Window's own tools, you'll learn the easiest way to write applications for Windows.

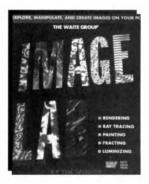
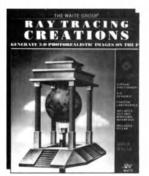


IMAGE LAB

Tim Wegner

This book is a complete PC-based "digital darkroom" that covers virtually all areas of graphic processing and manipulation. It comes with the finest graphics shareware available today: Piclab, CShow, Improces, Image Alchemy, and POV-Ray. This treasure chest of software lets you paint, draw, render and size images, remove colors, adjust palettes, combine, crop, transform, ray trace, and convert from one graphics file to another. Supercharged tutorials show how to make 3-D fractals and combine them to make photorealistic scenes. Plus, all the tools included have support forums on CompuServe so you can easily get the latest help and news.

ISBN 1-878739-11-5, 459 pages, 1-3.5" disk and color poster S39.95 Available Now



RAY TRACING CREATIONS

Drew Wells and Chris Young

With the Ray Tracing Creations book/disk combination, you can immediately begin rendering perfect graphic objects with ease. Using the powerful shareware program POV-Ray, you'll learn to control the location, shape, light, shading, and surface texture of all kinks of 3-D objects. POV-Ray's C-like language is used to describe simple objects, planes, shperes, and more complex polygons. Over 100 incredible pre-built scenes are included that can be generated, studied, and modified in any way you choose. This book provides a complete course in the fundamentals of ray tracing that will challenge and entice you. VGA display required.

ISBN 1-878739-27-1, 600 pages, disk and color plate section \$29.95 Available Now

TO ORDER TOLL FREE CALL 1-800-368-9369

TELEPHONE 415-924-2575 • FAX 415-924-2576

OR SEND ORDER FORM TO: WAITE GROUP PRESS, 200 TAMAL PLAZA, CORTE MADERA, CA 94925

Qty	Book	US/Can Price	Total	Ship to	
	Flights of Fantasy	\$34.95/48.95		Name	
	Fractals for Windows	\$34.95/48.95			
	Image Lab	\$39.95/55.95		Company	
	Lafore's Windows Programming Made Easy	\$29.95/41.95			
	Making Movies on Your PC	\$34.95/48.95		Address	
	Morphing on Your PC	\$34.95/48.95		City, State, Zip	
	Multimedia Creations	\$44.95/62.95		, , ,	
	Ray Tracing Creations	\$39.95/55.95		Phone	
	Virtual Reality Creations	\$34.95/48.95			
		\$36.95/51.95		ALL ORDERS MUST BE PREPAID	
Calif. re	Visual Basic SuperBible, Second Edition Walkthroughs and Flybys CD sidents add 7.25% Sales Tax	\$39.95/55.95 \$29.95/41.95		Payment Method ☐ Check Enclosed ☐ VISA ☐ MasterCard	
Shippin	g			Card#	Exp. Date
UPS Two	5 first book/S1 each add'l) Day (S10/S2) (S10/S4)			Signature	
	AT 1 77 T 18	TOTAL		SATISFACTION GUARANTEED	
				OR YOUR MONEY BACK.	

This is a legal agreement between you, the end user and purchaser, and The Waite Group®, Inc., and the authors of the programs contained in the disk. By opening the sealed disk package, you are agreeing to be bound by the terms of this Agreement. If you do not agree with the terms of this Agreement, promptly return the unopened disk package and the accompanying items (including the related book and other written material) to the place you obtained them for a refund.

SOFTWARE LICENSE

- 1. The Waite Group, Inc. grants you the right to use one copy of the enclosed software programs (the programs) on a single computer system (whether a single CPU, part of a licensed network, or a terminal connected to a single CPU). Each concurrent user of the program must have exclusive use of the related Waite Group, Inc. written materials.
- 2. Each of the programs, including the copyrights in each program, is owned by the respective author and the copyright in the entire work is owned by The Waite Group, Inc. and they are, therefore, protected under the copyright laws of the United States and other nations, under international treaties. You may make only one copy of the disk containing the programs exclusively for backup or archival purposes, or you may transfer the programs to one hard disk drive, using the original for backup or archival purposes. You may make no other copies of the programs, and you may make no copies of all or any part of the related Waite Group, Inc. written materials.
- 3. You may not rent or lease the programs, but you may transfer ownership of the programs and related written materials (including any and all updates and earlier versions) if you keep no copies of either, and if you make sure the transferee agrees to the terms of this license.
- 4. You may not decompile, reverse engineer, disassemble, copy, create a derivative work, or otherwise use the programs except as stated in this Agreement.

GOVERNING LAW

This Agreement is governed by the laws of the State of California.

LIMITED WARRANTY

The following warranties shall be effective for 90 days from the date of purchase: (i) The Waite Group, Inc. warrants the enclosed disk to be free of defects in materials and workmanship under normal use; and (ii) The Waite Group, Inc. warrants that the programs, unless modified by the purchaser, will substantially perform the functions described in the documentation provided by The Waite Group, Inc. when operated on the designated hardware and operating system. The Waite Group, Inc. does not warrant that the programs will meet purchaser's requirements or that operation of a program will be uninterrupted or error-free. The program warranty does not cover any program that has been altered or changed in any way by anyone other than The Waite Group, Inc. The Waite Group, Inc. is not responsible for problems caused by changes in the operating characteristics of computer hardware or computer operating systems that are made after the release of the programs, nor for problems in the interaction of the programs with each other or other software.

THESE WARRANTIES ARE EXCLUSIVE AND IN LIEU OF ALL OTHER WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR OF ANY OTHER WARRANTY, WHETHER EXPRESS OR IMPLIED.

EXCLUSIVE REMEDY

The Waite Group, Inc. will replace any defective disk without charge if the defective disk is returned to The Waite Group, Inc. within 90 days from date of purchase.

This is Purchaser's sole and exclusive remedy for any breach of warranty or claim for contract, tort, or damages.

LIMITATION OF LIABILITY

THE WAITE GROUP, INC. AND THE AUTHORS OF THE PROGRAMS SHALL NOT IN ANY CASE BE LIABLE FOR SPECIAL, INCIDENTAL, CONSEQUENTIAL, INDIRECT, OR OTHER SIMILAR DAMAGES ARISING FROM ANY BREACH OF THESE WARRANTIES EVEN IF THE WAITE GROUP, INC. OR ITS AGENT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THE LIABILITY FOR DAMAGES OF THE WAITE GROUP, INC. AND THE AUTHORS OF THE PROGRAMS UNDER THIS AGREEMENT SHALL IN NO EVENT EXCEED THE PURCHASE PRICE PAID.

COMPLETE AGREEMENT

This Agreement constitutes the complete agreement between The Waite Group, Inc. and the authors of the programs, and you, the purchaser.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above exclusions or limitations may not apply to you. This limited warranty gives you specific legal rights; you may have others, which vary from state to state.

VAITE GROUP PRESS"

SATISFACTION REPORT CARD

Please fill out this card if you want to know of future updates to Animation How-To CD, or to receive our catalog.

Company Name:						
Division/Departme	ent:		Mail Stop:			
Last Name:			First Name:		Middle Initial:	
Street Address:						
City:			State:		Zip:	
Daytime telephone	e: ()					
Date product was	acquired: Month	Day	Year	Your Occupation	ı:	
Overall, how would	you rate Animation How-	To CD?	Where d	id you buy this book?		
☐ Excellent	☐ Very Good	\square Good	☐ Book	store (name):		
□ Fair	☐ Below Average	□ Poor	☐ Disco	ount store (name):		
What did you like M	NOST about this book?					
				log (name):		
			□ Direc	ct from WGP 🗆 Oth	er	
What did you like LEAST about this book?			- What pri	What price did you pay for this book?		
				luenced your purchase of mmendation	this book?	
Please describe any	, problems you may have	encountered with	□ Maga	azine review	☐ Store display	
the Animation How	r-To CD		_ Mail	ing	☐ Book's format	
			□ Repu	tation of Waite Group	Press□ Other	
How did you use this	s book (problem-solver, tut	orial reference)	- How ma	ny computer books do yo	ou buy each year?	
,	s neek (pronioni sorroi) to	orial, rotoronco,		How many other Waite Group books do you own?		
			- What is	What is your favorite Waite Group book?		
□ New □ □ Power User □	of computer expertise? Dabbler	rienced Profession	is there	any program or subject y	rou would like to see Waite	
	jouges are you rummar wi		_ Group Fr	ress cover in a similar up	prouci:	
Please describe you	r computer hardware:		Addition	al comments?		
-	Hard disk_		=			
	3.5" disk dr					
	Monitor _					
	Peripherals					
Sound Board	CD ROM_		Chack	here for a free Waite G	roup catalog	

PLACE STAMP HERE

Waite Group Press, Inc.

Attention: Animation How-To CD

200 Tamal Plaza

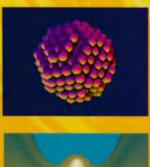
Corte Madera, CA 94925

----- FOLD HERE -----













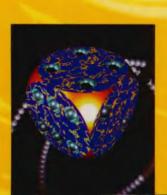


Animation How-To CD

Requires some familiarity with the basics of programming and graphics, but you won't need an advanced degree in Computer Science or Mathematics. You'll find all the powerful tools you need to turn your ideas into reality:

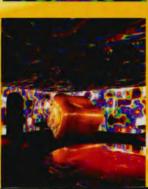
- The Polyray Ray Tracer creates photorealistic images and provides animation support, fast wire frame previews, and complete scene control.
- Dave's TGA Animator (DTA) converts ray traced images into flics and lets you create a wide variety of motions and effects.











THE WAITE GROUP®

ANIMATION HOW-TO CD







Flocks of skydiving frogs, columns of marching pencils, elastic diamonds dancing the lambada...create just about anything you can dream up with *Animation How-To CD* – your travel guide to the fantastic world of computer animation.

This intermediate-level book offers hands-on explanations that show you how to create dynamic moving objects. Each animation idea is explained, then the steps required to produce it are presented. Learn how to create belly-dancing logos, a disco full of gyrating dice, or a volcano that spews malted milk balls. Projects are based on the incredibly friendly ray tracer *Polyray*, and examples cover every variable that defines a scene. You'll even find over 450MB of finished animations, the complete source code, which can be adapted for your own flics, and tools on the bundled CD-ROM.

Discover the concepts of scene creation and object coordination: vantage point control, lighting, textures, object interactions, surface deformations, and motions based on real physical laws...or make up your own rules. Animation How-To CD lets you plunge into your imagination.



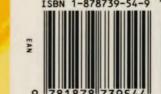


Operating System: MS-DOS

Recommended: Mouse, SVGA



WAITE GROUP PRESS™
200 Tamal Plaza
Corte Madera, CA 94925



\$34.95 USA \$48.95 CANADA